SPARK 1.0

User's Manual

Simulation Problem Analysis and Research Kernel

Lawrence Berkeley National Laboratory
Ayres Sowell Associates, Inc.

Contents

Section 1 Introduction

1.1 What is SPARK?

Simulation of a physical system requires development of a mathematical model, usually composed of differential and/or algebraic equations. These equations then must be solved at each point in time over some interval of interest. The Simulation Problem Analysis and Research Kernel (SPARK) is an object oriented software system to perform such simulations. By *object oriented* we mean that components and subsystems are modeled as objects that can be interconnected to specify the model of the entire system. Often the same component and subsystem models can be used in many different system models, saving the work of redevelopment.

1.2 Kinds of Problems

Since nearly any physical or biological system can be described in terms of a mathematical model, SPARK can be used in many scientific and engineering fields.

SPARK may be thought of as a general differential/algebraic solver. This means that it can be used to solve any kind of mathematical problem described in terms of a set of differential and algebraic equations. The term "continuous system" is often used to describe this class of problems. Typical examples include building heating and cooling systems, heat transfer analysis, and biological processes.

While, in principle, any system can be described in terms of differential and algebraic equations, there are many systems that are more easily described in terms of discrete states. Typical examples include assembly lines from the field of manufacturing engineering and queuing problems from various fields. SPARK is not designed for discrete state simulation problems. However, there are limited facilities for handling discrete events in otherwise continuous systems.

1.3 Describing Problems for SPARK Solution

Describing a problem for SPARK solution begins by breaking it down in an object oriented way (Nierstrasz 1989). This just means to think about the problem in terms of its components, with each component to be

represented by a SPARK object. Then, a model is developed for each component not already present in a SPARK library. Since there may be several components of the same kind, SPARK object models, i.e., equations or groups of equations, are defined in a generic manner, called *classes*. Classes serve as templates for creating any number of like objects that may be needed in a problem. The problem model is then completed by linking objects together, thus indicating how they interact, and specifying data values that specialize the model to represent the actual problem to be solved, and provide boundary values. Section 2.2 has several examples (See page 6).

Naturally, model descriptions must be expressed in some formal way. SPARK object class models are described in a textual language that is similar to other simulation programming languages except that it is non-procedural. That is, it is not necessary to order the equations, or to express them as assignment statements. This property derives from the input/output free manner in which the object classes are defined, and the use of mathematical graphs (McHugh 1990) to find an appropriate solution sequence.

In SPARK, the smallest programming element is a class consisting of an individual equation, called an *atomic class*. Then, *macro classes* bring together several atomic classes (and possibly other macro classes) into a higher level unit. Problem models are similarly described, using the atomic and macro classes, and placed in a *problem specification* file. When the problem is processed by SPARK, the problem specification file is converted to a *C++* program, which gets compiled, linked and executed to solve the problem for given boundary conditions.

You must have access to a *C++* compiler on the machine running SPARK. On Windows 95/98/NT platforms, the default WinSPARK installation assumes that you have Microsoft Visual *C++* installed, but Borland, GNU, and Symantec compilers are also supported. *VisualSPARK* on Windows 95/98/NT platforms normally use the MINGW implementation of the GNU C++ compiler, although the Cygwin implementation has also been used. UNIX installations normally use the GNU compiler, but SPARK has also been used with other compilers commonly available on Sun workstations.

While specifying problems in the SPARK language using existing classes is relatively easy, writing SPARK class models can be tedious. One necessary task is deriving the *inverses* for the class equation, i.e., closed-form solutions for several or all variables that occur in the equation. The labor of this task is multiplied in certain kinds of problems, such as those described in terms of partial differential equations. Such equations have to first be expressed as sets of ordinary differential equations, replicated many times with slight variations. To simplify these tasks, SPARK can be installed with symbolic tools, such as Maple (Char, Geddes et al. 1985). With such tools the user need specify only the atomic class equation, from which all necessary inverses and supporting *C++* functions are generated automatically through symbolic manipulation. For users without Maple, SPARK comes with its own symbolic manipulation tool that, while very limited, can find inverses of many equations encountered in simulation practice. For more involved problems, these symbolic tools offer a significant improvement in productivity. However, initially it will be more instructive for you to use SPARK directly, as we show in this Manual.

1.4 Portability and User Interfaces

SPARK is intended to be portable. The basic elements, i.e., the parser, setup program, and fixed elements of the solver, will compile and run on nearly any platform for which there is a *C++* compiler. In the initial release, executables, necessary source code, and graphical user interfaces are provided for the UNIX and Windows 95/98/NT platforms. On both platforms, the graphical user interfaces allow text-based creation of classes and problems using the SPARK language, as well as problem execution and results display. Post processing for visualization of results is supported in both environments.

This *User's Manual* is intended to cover the basic principles of SPARK programming. To the extent possible, it is intended to be independent of the platform. Consequently, examples are demonstrated using the command line interface only. Separate *Installation & Usage Guides* provide instructions for the individual platforms.

1.5 The History of SPARK

Although a general tool, SPARK was developed for use in simulation of building service systems, e.g., heating and air-conditioning. Most usage up to the time of this writing has been on systems from this field.

The first implementation of SPARK, which solved only algebraic problems, was done at the Lawrence Berkeley National Laboratory in 1986 (Anderson 1986). The basic ideas, including the graph-theoretic aspects, were based on earlier work at the IBM Los Angeles Scientific Center (Sowell, Taghavi et al. 1984). Buhl and Sowell extended the LBNL implementation to allow solution of differential equations in 1988 (Sowell and Buhl 1988). The MACSYMA and Maple interfaces were developed by Nataf and Winkelmann (Nataf and Winkelmann 1992), who also made many other improvements. Since that time, there have been new developments. For example, the solver was revised to decompose the problem into separately solvable components (Buhl, Erdem et al. 1993). Then in preparation for the initial public release, SPARK was completely rewritten in 1995-96. In this rewrite a new class and problem description language was implemented to improve modeling flexibility, and the solver was redesigned to improve solution speed. In addition, several user interface tools were developed, including a simple symbolic manipulation tool.

Section 2 Examples

2.1 Overview and Terminology

In this section we develop the main ideas and demonstrate the usage of SPARK in a tutorial manner. Mathematical problems are used for initial simplicity. Later sections extend these ideas to treat models based on actual physical systems.

We begin by defining some terminology. The basic entity in a SPARK model is the *object* that consists of a single algebraic equation and its interface or *port* variables. Objects are created by reference to a *class*, which may be thought of as a template for the equation object. As an example, consider the simple equation for the sum of two real numbers:

$$a + b = c \tag{2.1}$$

The *class* that we might call **sum** would contain this equation, and its *ports* would consist of the variables *a*, *b*, and *c*. Figure 2.1 is a pictorial representation of this idea.

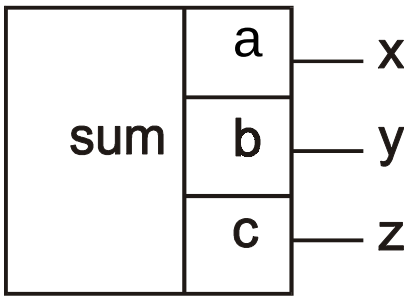

Figure 2.1 Sum Class Diagram

Note that we distinguish between an *object* and the *class* from which it is created. This is because there might be need for more than one equation of this form in a particular model. We can create as many instances (*objects*) from the *class* **sum** as we wish. Moreover, classes are saved, allowing their use in many different problems. In this way, SPARK reduces the model development work through code reuse.

Note also that the possibility of multiple instances of a class means that we must distinguish between the symbols used in defining the class and the corresponding variable names occurring in the problem definition. That is, if we wish to have the **sum** class represent both $x + y = z$ and $r + s = t$, it is obvious that

a must represent *x* in one place and *r* in another. We call variables such as *x* and *r problem variables* because they relate to a particular problem being described. On the other hand, *a*, *b*, and *c* relate only to the class definition and are called *interface* or *port* variables. It is also common to refer to SPARK problem variables as *links* because the keyword *link* is used to connect object ports, thus introducing the variable and assigning to it a name. We will see this in examples below.

2.2 Simple Math Problems

Although SPARK is intended for the analysis of complex physical systems represented as large systems of nonlinear equations, both algebraic and differential, an understanding of the basic methodology can best be obtained by working first with simple mathematical problems. We begin with the simplest possible problem, a single linear equation. This problem is then extended in steps to demonstrate more and more SPARK features. This will prepare us for dealing with more complex systems in later sections.

2.2.1 A Single Object Example

As a first exercise we will develop a SPARK solution for a simple math problem called *2sum*. In *2sum* we seek solutions for the equation:

$$x + y = z \tag{2.2}$$

As we saw in Section 2.1, there is a class in the SPARK foundation class library *globalclass* called **sum** which we can use to solve this problem. As shown in Figure 2.1 (See page 5), its *port* variables are *a*, *b*, and *c*, and it enforces the relationship of Equation 2.1. Obviously, by associating *a* with *x*, *b* with *y*, and *c* with *z* we can represent Equation 2.2 with an object of the **sum** class.

Equation 2.2 is a mathematical model involving three variables and one equation. To create a well-posed problem, we have to define two inputs. For this example, let's specify *x* and *y* as input, so that *z* is to be determined. The problem definition file *2sum.pr* then has the following contents:

```
/*   Problem Definition File
 *   for Simple Math Problem
 *        2sum.pr
 */
declare sum s;
input x s.a       report;
input y s.b       report;
link  z s.c       report;
```

Inputs are the quantities known at the outset. Links are variables to be solved for.

Here the *declare* statement creates an object **s** as an instance of the class **sum**. The *input* statements serve two functions. First, they associate the problem variables *x* and *y* with the corresponding object *port* variables **s.a** and **s.b** respectively. Note that we employ the notation **name.variable** to refer to the *port variable* of object *name*. Secondly, they indicate that these problem variables are inputs, as opposed to being determined by the solution process. Like *input*, the *link* statement associates problem variables with object *port* variables. However, links are variables to be solved for rather than inputs. The keyword *report* in *link* and *input* statements means that the variable should be reported in the SPARK output.

After creating *2sum.pr* as shown above, you must create an input file called *2sum.inp* with the following contents:

```
2       x       y
0       1       2
```

Here we see the format of a SPARK input file. The first line gives the number of input items, followed by their symbols as defined by the *input* statements in the problem specification file. The subsequent lines give values for each *input* variable, preceded by the time at which these values apply. If the problem is not a dynamic one, i.e., we are seeking a solution for only one set of inputs, only two lines are required as shown above. However, if we seek solutions at other time values, as many lines as needed can be given. This is discussed further when we take up dynamic problems in Section 2.6 (See page 22).

You can now run the problem with SPARK. The commands to do so differ somewhat depending on your platform. For a Windows 95/98/NT WinSPARK installation, type:

```
buildsolver 2sum.pr spark.prf <enter>
```

This results in creation of an executable program called *2sum.exe* Several other files are created, including *2sum.prf* and *2sum.run* which are needed to execute *2sum.exe.* To execute the program for numerical solution enter:

```
2sum 2sum.prf 2sum.run <enter>
```

If you are working with a UNIX or any *VisualSPARK* installation, the equivalent command is:

```
runspark <enter>
```

This builds and executes the single allowed problem file in the current working directory. It can be executed again without rebuilding with the command line:

```
2sum 2sum.prf 2sum.run <enter>
```

Since SPARK is often used to solve dynamic problems, run control information is needed when the program begins to execute. This information is provided in a problem run control file, *probName.run*, normally generated automatically when you first run a new SPARK problem. The file has the format of a SPARK preference file, discussed in Appendix B (See page 87).

The run control file for *2sum.pr*, i.e., *2sum.run*, is:

```
(
InitialTime             ( 0.0 ())
FinalTime               ( 0.0 ())
TimeIncrement           ( 1.0 ())
FirstReport             ( 0.0 ())
ReportCycle             ( 1.0 ())
DiagnosticLevel         ( 3 ())
InputFiles              ( 2sum.inp ())
OutputFile              ( 2sum.out ())
FinalSnapshotFile       ( 2sum.snap ())
InitialSnapshotFile     ( 2sum.init ())
)
```

The first five keys define the interval over which the problem is solved and other time related data. The *InitialTime*, *FinalTime*, and *TimeIncrement* control the solution interval and the closeness of the solution points in this interval. Since you may not wish to generate output at every solution point, you are allowed to specify when reporting is to begin and the interval between reporting with *FirstReport* and *ReportCycle* respectively. Because we are working a simple, algebraic problem here and we just want a single solution, we specify *FinalTime* to be the same as the *InitialTime* and *FirstReport* at time 0. *DiagnosticLevel* specifies the amount of intermediate output wanted. This is discussed further in Section 3.11.5 (See page 63). The remaining lines in the run control file specify various files related to the problem. We have already discussed the *2sum.inp* and *2sum.out* files. Here we see that in the *2sum.run* file you can specify where

these files are located in your directory structure. In the above example, they are specified to reside in the current working directory. The other two files, *FinalSnapshotFile* and *InitialSnapshotFile* are discussed in Section 3.13 (See page 64).

When the problem runs, summary output is displayed on the screen, and the principal output is written to a file called *2sum.out*. For this problem the *2sum.out* contains:

```
3       y       x       z
0       2       1       3
```

As with the input file, the first line gives the number of outputs, followed by the link names of each. The second line gives the time, followed by the result values for each output listed in the preceding line. As expected, adding 1 and 2 gives 3!

2.2.2 Arbitrary Input/Output Designation

With SPARK, the problem can be changed without changing the model.

The preceding example showed the basic steps required to set up a SPARK problem. However, it did not show SPARK's unique capabilities. One of these capabilities is that we can easily change which variables are input and which are output. That is, the problem can be changed without changing the model. For example, if we are interested instead in what *y* will be for specified values of *x* and *z*, we simply designate *z* as *input* and *y* as *link*:

```
/* Add 2 numbers together */
/*     2sum.pr            */
/*                        */
declare sum s;
input x s.a     report;
link  y s.b     report;
input z s.c     report;
```

And, we must also change the input file to be:

```
2       x       z
0       1       3
```

The resulting output file, *2sum.out*, contains:

```
3       z       x       y
0       3       1       2
```

Thus we see that *y* is calculated given *z* and *x*. Although shown here for a very simple problem with a single equation, this feature extends to more complex problems as well. The only requirement is that the model and the designated input variables must form a well-posed problem, i.e., one for which a solution exists.

2.2.3 Problems with Several Objects

The previous examples were problems with a single equation, thus requiring only one SPARK object. Most real problems involve more than one equation, and hence more than one object, raising the question of how objects are interconnected in SPARK. The two examples below show how this is done.

The problem we consider first is as follows:

$$\begin{cases} x_1 + x_2 = x_5 \\ x_3 + x_4 = x_6 \\ x_5 + x_6 = x_7 \end{cases} \tag{2.3}$$

Obviously, each of these equations can be represented by an object of class sum. The diagram in Figure 2.2 shows how these objects would have to be interconnected to represent this problem.

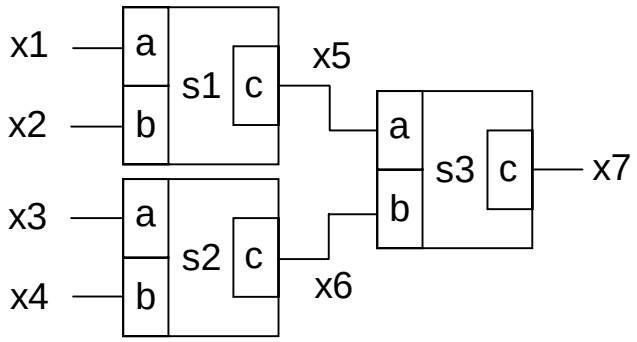

Figure 2.2 The 4sum example.

The problem specification file for this problem contains the following code:

```
/* Add 4 numbers together */
/*     4sum.pr            */
declare sum s1,s2,s3;
input x1 s1.a        report;
input x2 s1.b        report;
input x3 s2.a        report;
input x4 s2.b        report;
link  x5 s1.c, s3.a;
link  x6 s2.c, s3.b;
link  x7 s3.c        report;
```

Observe that the first *link* statement connects the problem variable *x5* to the *port c* of **s1** and *a* of **s3**, demonstrating the basic object interconnection method of SPARK. Any number of object ports can be specified following the problem variable name, causing all to be equated to the single problem variable defined in the *link* statement. The *link*, *input* and *declare* statements (plus a few others yet to be discussed) form the SPARK language. The complete language is presented in reference form in Section 4 (See page 71).

Because there are four *input* statements in *4sum.pr* there must be a *4sum.inp* file with values for the same four variables. This file is formatted as follows:

```
4       x1      x2      x3      x4
0       1       1       1       1
```

As before, the leading number in the first line, 4, is the number of inputs. It is followed by as many symbols, corresponding to *input* variables as defined in *4sum.pr*. The first number in the second line is the initial time, followed by values for each of the input variables.

The problem is built and executed using the same commands as for our *2sum* example (See page 6). The results are placed in *4sum.out* which is formatted like the input file:

```
5       x4      x3      x2      x1      x7
0       1       1       1       1       4
```

Several other files of interest are also produced when a SPARK problem is built and executed. First, various files with the extension *.log* may appear in the workspace. As you might suspect, these contain any error messages that may have been produced, as well as intermediate output from the numerical solution step.

Also produced is the equations file, e.g., *4sum.eqs*. For complex problems exhibiting numerical difficulties, it is sometimes useful to examine this file because it contains the computation sequence determined by SPARK and used to solve the problem. For *4sum* this file contains:

```
Inputs:
        x4
        x3
        x2
        x1
Component 0:
        x6 = sum(x3, x4)
        x5 = sum(x1, x2)
        x7 = sum(x5, x6)
```

In this file, inputs are listed first, followed by a sequence of assignments to problem variables, each computed by a right-hand-side function reference. These functions represent the inverses of the underlying class equation. In this case there is only one component, and it contains three function references in a non-iterative sequence. Later, we will see that in more complex problems SPARK will break problems down into several components that can be solved independently. Note that here we use the word "component" in a graph theoretic sense, meaning a group of nodes and edges, i.e., equations and variables, that can be solved together; these have nothing to do with physical components. Some components are *strongly connected*, meaning that there are cycles in that part of the graph. The practical significance of strongly connected components is that the corresponding equations have to be solved simultaneously, by iteration.

As with the single object example, we can use the same model to solve different problems by changing what is input and what is solved for. For example, suppose we want to specify *x5* and determine *x1*. The problem file is then:

```
/* Add 4 numbers together */
/*     4sum.pr            */
/*                        */
declare sum s1, s2, s3;
link  x1 s1.a          report;
input x2 s1.b          report;
input x3 s2.a          report;
input x4 s2.b          report;
input x5 s1.c, s3.a;
link  x6 s2.c, s3.b;
link  x7 s3.c          report;
```

A suitable input file is:

```
4       x5      x2      x3      x4

0       2       1       1       1
```

After building and executing, the resulting 4*sum.out* file is:

```
5       x4      x3      x2      x7      x1

0       1       1       1       4       1
```

And the equations file shows the solution sequence:

```
Inputs:
        x4
        x3
        x5
        x2
Component 0:
        x6 = sum(x3, x4)
        x7 = sum(x5, x6)
        x1 = difference(x5, x2)
```

Just as you might do, based on Figure 2.2 (See page 9) SPARK evaluates **s2** followed by **s3** in the "forward" direction yielding *x6* and *x7*, then evaluates **s1** in the "reverse" direction to get *x1*.

2.2.4 Problems Requiring Iterative Solution

Up to this point all of our examples have been such that non-iterative solutions could be found. In more complex problems this may not be possible. For example, consider the set of equations below, in which c_1 and c_2 are given and x_1, x_2, x_3 and x_4 are to be determined.

$$\begin{cases} x_1 + x_3 + x_2^2 + \sqrt{x_2} = c_1 \\ x_2 = x_1 e^{x_1} \\ x_1 x_4 + x_3 x_4 + x_4^3 = c_2 \\ x_4 = x_3 e^{-x_3} \end{cases} \tag{2.4}$$

This set of equations does not have a closed form solution, and is very difficult to solve by any means. In fact, with some values of c_1 and c_2, it has no solution at all. However, with $c_1 = 3000$ and $c_2 = 1$ there is a solution and SPARK can easily find it.

The problem can be specified for SPARK exactly as for simpler ones. Figure 2.3 shows a SPARK diagram with objects and interconnections.

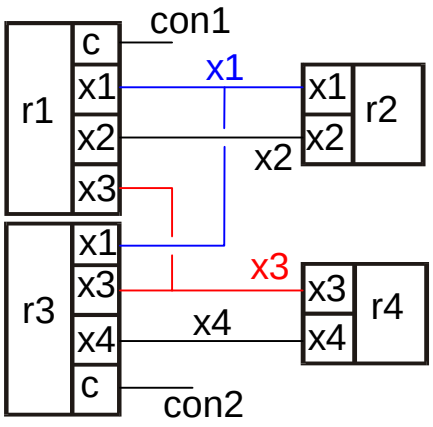

Figure 2.3 Four nonlinear equations.

In this case we have used four objects, each representing one of the equations. We assume for the moment that there are classes **r1**, **r2**, **r3**, and **r4** representing the equations in the order given previously, presumed

to have been defined and placed in the class directory.[1] The SPARK problem file can then be constructed as follows:

```
/*  Four nonlinear equations */
/*       example.pr          */
declare r1  r1 ;
declare r2  r2 ;
declare r3  r3 ;
declare r4  r4 ;
input con1  r1.c                                  report;
input con2  r3.c                                  report;
link x1     r1.x1 match_level = 0, r2.x1, r3.x1   report;
link x2     r1.x2, r2.x2                          report;
link x3     r1.x3, r3.x3, r4.x3                   report;
link x4     r3.x4, r4.x4                          report ;
```

The two constants, c_1 and c_2 in the equations, are defined as *inputs* **con1** and **con2**. In these *input* statements, note that the port variables representing the c_1 and c_2 constants are called *c* in $\mathbf{r_1}$ and $\mathbf{r_3}$. Similarly, in the *link* statements it is evident that the other port variables have the same names as the corresponding problem variables. Normally, in the interest of code reuse, it is better to define a generic class using local names for port variables, as we have done in the earlier examples. Here, however, where it is unlikely that we will have need for other instances of these rather specialized objects, it would introduce unnecessary confusion to employ different port and problem variable names. Hence the $\mathbf{x_1}$ problem variable is linked to the $\mathbf{x_1}$ port variable of all objects in which it occurs, i.e., $\mathbf{r_1}$, $\mathbf{r_2}$, and $\mathbf{r_3}$.

A new SPARK language keyword, *match_level*, is used in this problem. The purpose of this keyword is to provide a hint to SPARK on how to match certain variables to certain equations. Here, by placing the *match_level* = 0 after the **r1** *port* connection for **x1** we are discouraging SPARK from using the **r1** object, i.e., the first equation, to calculate **x1**. Although most often SPARK can do without such hints, there may be times when you have particular insights into the numerical properties of the problem, and the *match_level* keyword provides one mechanism for capitalizing on this knowledge. For example, experience with the current problem indicated that the above *match_level* restriction leads to a better solution sequence. Unfortunately, it is not always easy to discover appropriate matching preferences, but when you do develop the insight for a particular problem it is important to be able to control SPARK in this manner. This subject is discussed further in Section 3.11.3 (See page 60).

The results of running SPARK on the problem so described, with values of 3000 and 1 for the constants c_1 and c_2, respectively, are shown below:

6	con2	con1	x4	x1	x2	x3
0	1	3000	0.288576	2.9273	54.6738	0.454716

Naturally, the values reported for x_1 through x_4 satisfy the given equations.

The equations file, *example.eqs*, shows how SPARK arrived at these answers:[2]

[1] We will see how to define SPARK object classes in Section 2.4 (See page 14).

[2] As is often the case for nonlinear problems, this example has multiple solutions. The solution found will depend upon the starting point in the iterative solution process.

```
Inputs:
        con2
        con1
Component 0:
        x4 = r4_b(x3)
        x1 = r3_a(x3, x4, con2)
        x2 = r2_b(x1)
[break] x3 = r1_c(x1, x2, con1)
           = x3 [predictor]
```

We see there is a single component (called "Component 0") in the solution, meaning that this problem does not allow partitioning. Within this single component, the function *r4_b(x3)* represents object *r4*, i.e., the fourth equation in (2.4), solved for *x4* in terms of *x3*. The value returned by the function is assigned to the *x4* problem variable. Similarly, *r3_a(x3, x4, con2)* represents object *r3*, i.e., the third equation, solved for *x1*, *r2_b(x1)* is *r2* solved for *x2*, and finally *r1_c(x1, x2, con1)* is *r1* solved for *x3*. It is apparent that these assignments form a cycle, i.e., *x1* must be known to get *x3*, but *x3* must be known to get *x1*. That is, the component is strongly connected. Recognizing this, SPARK has selected *x3* to break the cycle, i.e., a value of *x3* is guessed to start an iterative solution process. Thus after evaluating *r1_c* (using the guessed value of *x3* to get *x4* and then *x1* and *x2*) SPARK will use a prediction method for estimating a new value of *x3* and repeat the calculations from the first assignment. This will continue until the predicted and calculated values of *x3* agree to within the SPARK precision, which defaults to 10^{-6}. At first, prediction is done with the Newton-Raphson method. If convergence is not achieved, alternate methods can be tried. Usually, convergence is obtained with the Newton-Raphson method.

The above functions are based on the respective object class equations. By chance, *r4_b* happens to be the way the *r4* equation was originally expressed, i.e., as a formula for *x4* in terms of *x3*. However, *r3_a* is the *r3* object class rearranged symbolically, i.e.,

$$x_1 = (c_2 - x_3 x_4 - x_4^3) / x_4$$

This is called an *inverse* of the object. Part of the task of developing a SPARK class is performing these symbolic inversions of the given equations, and embedding them in *C++* functions. This is discussed in Section 2.4.2 (See page 16).

2.3 Well Posed Problems

In Section 2.2.2 (See Page 8) we saw that SPARK allows us to change which problem variables are input and which are to be solved for without changing the underlying model. This flexibility is the result of specifying object models without *a priori* specification of inputs and outputs (Sahlin and Sowell 1989). Thus we were able to solve for x_5, x_6, and x_7 in the example Equation 2.3 (See Page 8) given x_1 through x_4, or by a simple change of *input* and *link* designations solve for x_1, x_6, and x_7 given x_2, x_3, x_4, and x_5.

It would be grand if we could say that this selection of the input and output sets was completely arbitrary. For example, in the example of Section 2.2.3, Equation 2.3 (See page 8), there are 3 equations (objects) and 7 variables, so one might hope that any set of 4 *input*s could be used to determine the remaining 3 variables. However, we are constrained by what is mathematically possible. In many problems there are sets of inputs that will not define a problem that has a solution. For example, if we specified x_2, x_3, x_4, and x_6 it is impossible to determine a solution. From Figure 2.2 we see that if x_3 and x_4 are both specified then x_6 *cannot* be specified. Moreover, there is no way to determine x_1, x_5, and x_7 given only x_6. Mathematically, a problem is said to be *well posed* if it admits a solution. Thus with this input set we have an ill posed problem.

Naturally, SPARK has no ability to solve ill-posed problems.[3] In the case here, SPARK can immediately determine that the problem is not well posed; specifically, it discovers that there is no possible *matching* of equations and variables. Other forms of ill posedness cannot be discovered until a numerical solution is attempted. In such cases a lack of convergence will be reported. Unfortunately, however, lack of convergence also may be the result of other numerical problems, such as improper starting values, so we cannot always conclude that this means ill posedness. Problems of this nature are all too familiar to those who routinely work with nonlinear systems of equations. Often, insights afforded by knowledge of the physical problem under analysis suggest ways to fix the numerical problem. In seeking to resolve these difficulties, we should be motivated by the realization that proper mathematical models of physical systems are well posed. Otherwise, the physical system could not behave in the observed way.

In summary, SPARK offers a method for specifying and solving sets of equations, provided solution is possible. But it should be no surprise that it cannot solve insoluble problems, and numerical difficulties may be encountered as they would be in other solution methods.

2.4 Creating SPARK Atomic Classes

The examples so far have made use of existing SPARK object classes. In practice, it is often necessary to create new object classes to meet special needs. This can be done either by hand, or with symbolic tools such as the SPARK symbolic solver, or third-party tools like *Maple*, *Mathematica* or *MACSYMA*. Here we will see the manual process. This will allow you to better understand the use of the symbolic tools, as discussed in Section 3.8 (See page 49).

2.4.1 Class Definition

Creating a SPARK object is a two step process. First, you must create the object class definition. Second, the inverse functions required by the class must be expressed in *C++* following the SPARK function protocol. The class definition and the supporting *C++* inverse functions are stored in the same file with a *.cc* extension. These steps are demonstrated below for the **sum** atomic class.

```
/* SPARK sum class definition */
#ifdef SPARK_PARSER
PORT a      "Summand 1";
PORT b      "Summand 2";
PORT c      "Sum";
EQUATIONS {
      c = a + b;
}
FUNCTIONS {
      a = difference( c, b );
      b = difference( c, a );
      c = sum( a, b );
}
#endif /* SPARK_PARSER */
#include "spark.h"
/* "difference" inverse function*/
#define     a     args[0]
#define     b     args[1]
#define     c     result
```

[3] Indeed, it is contradictory to even suggest it!

```
double difference(ArgList args)
{
      double      result;
      c = a - b;
      return result;
}
#undef a
#undef b
#undef c
/* "sum" inverse function*/
#define     a     args[0]
#define     b     args[1]
#define     c     result
double sum( ArgList args )
{
      double      result;
      c = a + b;
      return result;
}
#undef a
#undef b
#undef c
```

As shown above, it is customary to begin a class with comments, to describe what it does. After the comment header comes the body of the class definition. This is placed within C-style *#ifdef* and *#endif* so the file can be processed both by the SPARK *parser* and the *C++* compiler.

The first part of the class definition is a list of the ports. It is through these ports that objects of the class communicate with other objects. Although the *port* statement has additional optional clauses, the only required part is the name of the port variable. Here, we also provide a description string that is used for error reporting. The port variable name can be arbitrarily chosen and of any length and is placed following the *port* keyword. Note that throughout the SPARK language user selected names are case sensitive. However, keywords of the language are not. Thus either *port* or *PORT* will do, but *a* and *A* are considered different *port* names. Like all SPARK statements, the *port* statement can span multiple lines if necessary. Each *port* statement ends with a semicolon.

After the *port* declaration, the equation for the class can be given in the optional *equations* block. Although SPARK atomic classes presently have a single equation, the possibility of multiple equations is allowed for with the compound statement using braces, *equations {…}*.[4]

Following the equations is the *functions {..}* compound statement. A function for calculating each port variable can be given between the braces. Here we define functions for calculating each of the three port variables. Normally, this is the best practice, since it allows SPARK greatest flexibility and efficiency in devising a solution strategy for various problems in which the class might be used. That is, some problems may require *c* to be determined in terms of *a* and *b*, while in others it may be preferred to calculate *b* given *a* and *c*. As we shall see below, each function is an *inverse* of the object equation.

For complex equations, some inverses may be difficult or impossible to obtain. Or, it may be that special knowledge about the problem under investigation suggests that a particular inverse should not be used, because, for example, it might lead to numerical difficulties. For these reasons, SPARK allows you to omit unavailable or unwanted inverses. For example, we could simply omit the function for calculating *a* from the **sum** class. Should the need to calculate *c* from *a* and *b* then arise in some problem using the class, SPARK would have to perform the calculation iteratively.

---

[4] The equations block is optional since SPARK currently does not process it. Future releases may automatically generate the *C++* functions based on the equation block.

2.4.2 Inverse Functions Definition

After the class definition comes the definition of the inverse functions. These functions, supporting the SPARK class definitions, are expressed as *C++* functions. Although some familiarity with *C++* would be helpful here, you should be able to understand the discussion with background in any similar language.

The basic structure of an inverse function in a SPARK atomic class is:

```
double funct_name( ArgList args )
{
      // Code for calculating the result from the arguments,
      // returned as a double.
}
```

The arguments must be passed as an array of type `ArgList`. However, it is customary to alias elements of this array to the symbols used in the equation. Also, the *result* is aliased to the symbol for the returned variable. This practice not only makes the functions easier to read (and write!), but also simplifies their automatic generation with symbolic tools. Note the `#include "spark.h"` which (indirectly) provides the definition of `ArgList`.[5]

In our **sum** example above we used the C preprocessor *#define* directive to alias function arguments. However, there are alternative ways to do this that take advantage of advanced *C++* language features, for example, reference variables and the *const* qualifier. A reference variable is declared with a preceding *&* and is initialized in the declaration, e.g.:

```
double &x = y;
```

meaning that the identifier *x* is merely an alias for a previously declared identifier *y*; there is no separate storage location for *x*. If *const* precedes the declaration, the reference variable cannot be changed from the initialized value. Consequently, a *const* reference variable is functionally equivalent to a *#define* constant.

With these features we can write the previous function definition for **sum,** omitting the *#define* 's, as follows:

```
double sum( ArgList args )
{
      const double &a = args[0];
      const double &b = args[1];
      double      result;
      double &c = result;
      c = a + b;
      return result;
}
```

This is functionally equivalent to the previous code, but has the advantage of allowing type checking by the compiler, and automatic type conversion when needed. The latter is important if the argument happens to be used as an actual argument to another function that is called within the SPARK function. When *#define* is employed in this case, most compilers will not be able to properly determine the function argument type.

Yet another way to express a SPARK inverse function in *C++* is similar to the above, but does not use reference variables on the arguments:

[5] More precisely, `ArgList` is defined as a pointer to the `TArgument` class in *value.h*.

```
double sum( ArgList args )
{
      const double a = args[0];
      const double b = args[1];
      double      result;
      double &c = result;
      c = a + b;
      return result;
}
```

The difference between these two styles is that in the latter there is local storage for the variables *a* and *b*, and the arguments are copied into these locations every time the function executes, while in the former there is no such storage or copying. The compiler will generate code that refers to the data where it is stored in the caller. Thus the former is in principle more efficient. However, cursory testing has failed to show significant empirical differences.

SPARK functions can be as simple as the above example, or quite complicated. The full expressive power of *C++* is allowed. Note also that code for existing models can be integrated by means of a function call. Furthermore, by following the rules for mixed language programming in your environment, the referenced functions can be in FORTRAN, Pascal, or assembly language. The principal requirement is that a single result must be returned.[6] Care should also be taken that *the function not depend upon retained state*, i.e., the value of a local variable from a previous call, since a SPARK problem may instantiate more than one object using it. Perusal of some of classes in *vspark\globalclass* and *hvactk\class* directories may be beneficial before beginning development of complex classes of your own.

2.5 Models of Physical Systems

The previous examples were purely mathematical in nature. They allowed us to discuss the basic ideas in SPARK, unencumbered by details. Here we take up some of the other issues that arise when modeling physical systems. In particular, we show how SPARK handles the problem of unit consistency, and range of values for variables. Also, we show provision in SPARK for modeling at a level higher than individual equations. Then, using these new ideas, we show the development of a SPARK model for a system of modest complexity.

2.5.1 Units, Valid Range, and Initial Values

When simulating real physical systems, there must be consistency in the units of measure throughout the problem. In terms of a SPARK problem specification, this means that the units of a problem variable linked to an object *port* must be the same as the units assumed for the port variable when the object class was defined.

SPARK has a limited capability to ensure unit consistency. This is provided by associating an optional *unit string* with each *port*. Then the SPARK processor can check and report an error if you inadvertently connect variables of different units. Also, you can give initial, minimum, and maximum values for the port variable. For example, the *cpair.cc* class from the HVAC Toolkit has a *port* for the specific heat coded as follows:

```
port CpAir "Specific heat of air" [J/(kg_dryAir*deg_C)]
            init = 1.0  min = 0.01  max = 5000.0;
```

[6] Future releases of SPARK may allow multivalued objects, removing this restriction.

The unit string is placed in square brackets […]. Any connection to this *port* will have to have an identical units string. The *min* and *max* values have the obvious meaning; run-time warnings are issued when the value is outside this range. The *init* value is used whenever SPARK needs a starting value and none is provided elsewhere. For example, if the associated variable happens to be a break variable the very first iteration will use the *init* value of 1.0 for **CpAir**.

In order for SPARK units checking to work to your benefit you must define a consistent set of units. Table 2.1 shows the SI units used in the HVAC Toolkit (See Appendix A Using the HVAC Tool Kit on page 81). Other consistent sets could be used instead. Note that the units and value ranges given in Table 2.1 are not built into SPARK; they are simply the units employed in the HVAC Toolkit class library. However, they do serve as an example of a consistent set of units. When developing SPARK models you have the choice of adhering to these units, or developing your own library with whatever units you choose. Obviously, you should be consistent with whatever unit system you choose. Otherwise, you will have to implement special unit conversion objects when your objects are connected. The *init*, *min*, and *max* values should be set as appropriate for each *port*.

Table 2.1 ***SPARK Units (SI) used in the HVAC Toolkit.***

Unit String	Description	Initial	Minimum	Maximum
[-]	Unspecified	0.	-1000000.	1000000.
[J/kg_dryAir]	Enthalpy, air	25194.2	-50300.0	398412.5
[J/kg_water]	Enthalpy, water	25194.2	-50300.0	398412.5
[kg_water/kg_dryAir]	Humidity ratio	.002	0.0	0.1
[kg_dryAir/s]	Mass flow rate, air	10000.	1000.	1000000.
[kg_water/s]	Mass flow rate, water	10.	0.	1000.
[deg_C]	Dry bulb temperature	20.	-50.	95.
[m^3/kg]	Specific volume, fluid		1.0	
[m^3/kg_dryAir]	Specific volume, air	0.8332	0.6	1.6
[kg/m^3]	Ratio of total (air plus moisture) mass to volume	1.2026	0.6	1.8
[J/kg]	Enthalpy, steam			
[J/(kg*deg_C)]	Specific heat, fluid	1.0	0.01	5000.0
[J/(kg_dryAir*deg_C)]	Specific heat, air	1.0	0.01	5000.0
[kg/s]	Mass flow rate, fluid		0.0	
[m^3/s]	Volumetric flow rate, fluid		1	
[m]	Distance		1	

[m^2]	Surface area		0.	
[W]	Power	1	-10000	10000
[Pa]	Pressure	101325	0	110000
[W/deg_C]	U*A, heat transfer	0	-1.0E6	1.0E6
[s]	Time, seconds	0.0	0	1.0E30
[fraction]	Any ratio	1.0	0.0	1.0
[scalar]	Any non-dimensional	1.0	-1.0E30	1.0E30

To demonstrate, consider the *sercond.cc* class from the HVAC tool kit, which models two conductors in series. The ports are defined as:

```
PORT U1     "Conductance 1"              [W/deg_C];
PORT U2     "Conductance 2"              [W/deg_C];
PORT UTot   "Overall conductance"        [W/deg_C];
```

Then, when the **sercond** class is used in a problem definition you have to give matching unit strings at each *link* or *input* statement for the problem variables connected to the ports of **sercond**:

```
declare sercond  sc;
input UA1 sc.U1         [W/deg_C]   report;
input UA2 sc.U2         [W/deg_C]   report;
link  UATotal sc.UTot   [W/deg_C]   report;
```

The SPARK parser can then check to be sure you have not made a units error; if the units string in a *link* or *input* does not match those of all port variables in the same statement, a units error will be reported.

There are times when you may not want strict enforcement of unit consistency. For example, the **sum** object class is used in many places, sometimes adding heat flux and other times mass flow rates. If we insisted on strict unit consistency, we would have to have a separate **sum** class for every different case. To avoid this problem, and to allow for problems where units are not important, there is an unspecified unit identifier. Units on a *port* are unspecified when you do not give any unit information, or when you explicitly declare unspecified units with [-] as the unit identifier. When a *port* has unspecified units, no unit checking is done on links to that *port*.

2.5.2 Macro Objects

Because SPARK uses a computation graph based on individual problem variables and equations, the SPARK object must be a *single equation*. While this is an advantage for efficient solving, the disadvantage is the tedium of defining a large system model entirely in terms of individual equations. When modeling physical systems, it is sometimes more convenient to work in terms of larger elements, such as models of physical components or subsystems. Such models most often will involve several equations and variables rather than one.

We have already mentioned in Section 2.4.2 (See page 16) that one way to include more complex models is by placing the equations within the *C++* functions required by ordinary SPARK atomic classes. However, this is a very limited idea. One limitation is that only a single result can be communicated to the rest of the problem, even though many variables may be determined in the process. Another is that the user becomes

responsible for devising an algorithm for the function, thereby bypassing one of SPARK's most unique capabilities.

Macro classes let you to work at a high level of abstraction, while allowing SPARK to employ efficient, equation-based solution strategies.

The macro class provides a better mechanism for allowing more complex SPARK classes. It allows multiple atomic classes, and even other macro classes, to be assembled into a single entity for use by the model builder. Macro classes are used in problems or in other macro classes exactly like atomic classes, i.e., by use of the *declare* keyword. However, when processed by the SPARK parser, any declared macro objects are separated into atomic objects so that the graph-theoretic solution methods can be applied in the normal manner. This allows you to work at a high level of abstraction, while allowing SPARK to employ efficient, equation-based solution strategies.

As an example of the need for a macro class, consider the flow of air in a duct network, such as might occur in a heating system for a building. In simulation of systems like this there is a need for models of various components such as diverters that split the flow into two streams and mixers that merge the flow of two duct sections into one. Here, let's focus on the mixer and devise a model for it in the form of a SPARK macro class.

The diagram in Figure 2.4 shows the mixer component.

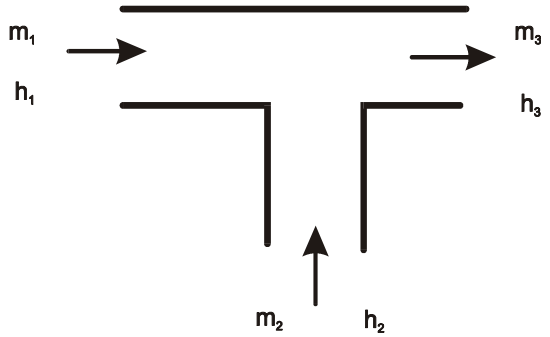

Figure 2.4 Dry air mixer.

The air duct mixer model must include two laws from physics: conservation of mass and conservation of energy. These can be expressed in the following equations:

$$\begin{cases} m_1 + m_2 = m_3 \\ m_1 h_1 + m_2 h_2 = m_3 h_3 \end{cases} \tag{2.5}$$

where *m* represents mass flow rate and *h* represents the enthalpy of the air streams. The subscripts *1* and *2* represent the conditions at the two inlets, and *3* that at the outlet.

To construct a macro object class for the mixer we shall assume that we already have object classes for the mass and energy balance equations. Actually, the mass equation can be represented with the familiar **sum** class. Also in the SPARK object library there is an object class called **balance** that represents equations like the enthalpy one. The port variables analogous to *m* and *h* are *m* and *q* respectively.

The *macro class* will connect the constituent classes exactly as if we were creating a problem definition file. Constituent class port variables that are to have the same meaning in the context of our new macro class are linked together, forcing equivalence. Those that are to be available for interfacing to problems or other macro classes are "elevated," i.e., made port variables of the macro class.

Figure 2.5 shows this idea and serves as a guide in writing the macro class. Because all represent the same quantity, the macro port variable *m1* must be connected to the **a** *port* of the **sum** class and the **m1** *port* of

the **balance** class. Other port variables are linked in a similar manner. The SPARK expression of this is shown below.

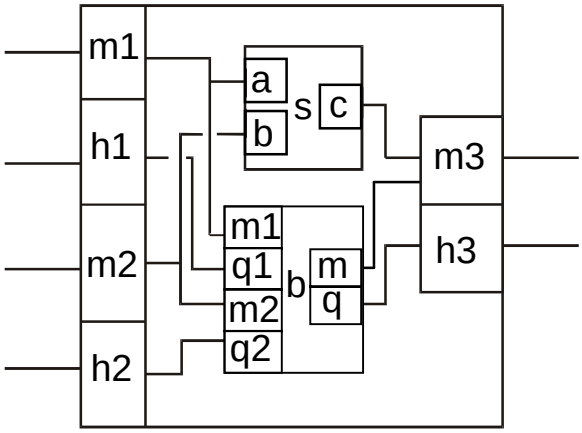

Figure 2.5 Mixer macro class diagram.

```
/*    SPARK Mixer Object Macro Class
 *
 */
port m1 "Stream 1 mass flow rate" [kg_dryAir/s];
port m2 "Stream 2 mass flow rate" [kg_dryAir/s];
port m3 "Stream 3 mass flow rate" [kg_dryAir/s];
port h1 "Stream 1 enthalpy" [J/kg_dryAir];
port h2 "Stream 2 enthalpy" [J/kg_dryAir];
port h3 "Stream 3 enthalpy" [J/kg_dryAir];
declare sum s;
declare balance b;
link  mass1 .m1, s.a, b.m1;
link  mass2 .m2, s.b, b.m2;
link  mass3 .m3, s.c, b.m;
link  enthalpy1   .h1, b.q1;
link  enthalpy2   .h2, b.q2;
link  enthalpy3   .h3, b.q;
```

It will be observed that this is very much like a problem definition. The principal difference is the absence of inputs. Also, note that a macro class has ports, whereas a problem does not. Ports provide the interface to the outside. That is, when an object of this class is used, connections will be made to its *ports*. The internal *links*, on the other hand, are not exposed to the outside at all. If you want a variable represented by a macro class *link* to be available for outside connections, you must connect it internally to a *port*. For example, the line:

```
link  mass1 .m1, s.a, b.m1;
```

means that the *link* named **mass1** connects the **m1** *port* of the mixer macro class to the a *port* of **s** and the **m1** *port* of **b**.

Note the dot (.) in front of the first connection following the *link* names in the above example. The rationale for the dot syntax is based on the general connection notation **x.p**, where we are referring to the **p**

port of **x** object. When the *port* in question belongs to the macro class being defined, as opposed to one of its constituents, the class name is that of the very class we are defining, and therefore is not expressed.[7]

The similarity between macro classes and problems makes it common practice when developing a macro class to first test it as a problem. For example, you could develop the mixer class as a problem, saving it in a file with *.pr* extension. Once it is working properly, you simply change the inputs to links, add ports for the variables needed at the interface, connect the corresponding links to these ports, and save it as a *.cm* file.

You may have noticed in the above example that the name of links, e.g., **mass1**, are not used anywhere. This is because we express the internal connections entirely in terms of the class and *port* names, as in **s.a**, or with an implied class name and *port* name as in **.m1**. Because *link* names are not used, they are optional when defining macro classes. That is, we could write:

```
link  .m1, s.a, b.m1;
```

instead of the previous statement with exactly the same effect. In contrast, link names are required for problems, as these are the names by which we know the problem variables. Further discussion of *link* names is provided in Section 3.6 (See page 47).

Note that we have included unit strings in the ports. This will prevent you from connecting inappropriate links to objects of the mixer class. Also, we could have placed unit strings in the links to allow unit checking of the links to the ports of the classes which are used in the macro. We elect not to do so here, however, because both **sum** and **balance** are mathematical classes with generic ports.

Finally, note that macro classes are entirely equivalent to normal SPARK classes in terms of usage. They can be used in creating problem specification files, or in building other macro classes. The SPARK parser recursively expands the macro objects as it generates the solver code.

2.6 Differential Equations

Thus far we have focused on problems with only algebraic equations. However, many simulation problems are dynamic in nature and involve differential equations as well. That is, some of the problem variables appear as derivatives with respect to time. In this Section we see that SPARK is capable of representing and solving such problems. We begin with a brief review of numerical methods used in solving ordinary differential equations.

2.6.1 Numerical Solution of Differential Equations

Numerical solution methods for differential equations start with given initial values of dynamic variables[8] and attempt to project to a new solution a short time later. When the differential equation is part of a larger system of equations the entire set must be solved at each point to ensure accuracy. The process is then repeated, with the newly calculated values becoming the basis for the next projection forward. The amount by which time is advanced at each projection is called the *time step*, referred to as *TimeIncrement* in the run control file. Generally speaking, the time step has to be small in order to achieve sufficient accuracy of the solution. Since simulations are often carried out over long periods of time, many small time steps are required. Computational efficiency is therefore very important.

[7] In some object oriented languages, such as C++, the name of the class being defined is known internally as *this*. In SPARK we chose to have the name *this* be understood rather than expressed.

[8] Here we shall call variables appearing in differential form *dynamic variables*

The projection is done by means of an *integration formula* involving current and/or past values of the variables and their derivatives. For example, the simple Euler integration formula *f* is:

$$x = f\left(x_p, \dot{x}_p\right) = x_p + h\dot{x}_p \tag{2.6}$$

where *x* is the dynamic variable, $\dot{x}$ is its derivative with respect to time, and *h* is the time step. Note that the Euler formula involves variable and derivative values only from the *previous* time, indicated by the subscript *p*. This is called an *explicit* formula because it gives the new solution explicitly, i.e., without reference to unknown values at the end of the time step. On the other hand, some integration formulas do involve values of the dynamic variables at the new time, i.e.,

$$x = f(x_p, \dot{x}_p, x, \dot{x}) \tag{2.7}$$

Such formulas are called *implicit* because they involve values at the new point as well as past values.[9] Obviously, iteration is required for implicit integration formulas, while not for explicit integration formulas. The aim of the more complex formulas is to get improved accuracy and numerical stability with larger time steps.

2.6.2 How SPARK Deals with Differential Equations

SPARK deals with differential equations by introducing object classes to represent integration formulas. These can be from the SPARK *globalclass* library, or user defined. You can define many different kinds of integration object classes, ranging from simple explicit formulas such as Euler's to complex implicit formulas used in predictor-corrector methods. Unlike other simulation languages, SPARK even allows you to use different integration formulas in different parts of the same problem.

Below we will learn how to solve a simple differential equation. We will first use integrators from the SPARK library, and then see how integrator object classes are created. In Section 2.7 (See page 28) this will be extended to a more complex problem with mixed algebraic and differential equations.

2.6.3 Solving a Simple Differential Equation

As a simple example, consider the differential equation:

$$\dot{x} + ax = b; \qquad x(0) = x_0 \tag{2.8}$$

where $\dot{x}$ is understood to be the derivative of *x* with respect to time, *t*, the independent variable. We see this to be a well-posed problem; given a, b, and x_0 it can be readily solved for $x(t)$.

To achieve a numerical solution in SPARK we view the derivative as a separate dependent variable. In order to preserve the balance between equations and variables, this additional variable requires an additional equation to be added to the set. An integration formula provides this needed equation, giving the value of *x* at the next point in time. If we employ the Euler formula, Equation 2.6, the set of equations to be solved is:

[9] The terms *open* and *closed* are sometimes used instead of explicit and implicit, respectively.

$$\begin{cases} \dot{x} + ax = b; & x(0) = x_0 \\ x = x_p + h\dot{x}_p \end{cases} \tag{2.9}$$

It is seen that we again have a well-posed problem, two equations in the two variables *x* and $\dot{x}$. Since both equations are algebraic, they can be easily solved by the established SPARK methodology.

This example is simple, but the method is general. Regardless of problem complexity, we simply introduce a new problem variable for every (first order) derivative, and at the same time introduce an integrator object for the dynamic variable.

The SPARK solver then has an algebraic problem to deal with. Observe also that implicit integration formulas require no special consideration. Such formulas involve the *x* at the new time, i.e., are implicit in *x*:

$$x = f(x_p, \dot{x}_p, x, \dot{x}) \tag{2.10}$$

But this is of no concern, because the SPARK solver anticipates that an iterative solution process may be necessary due to the possibility of other cycles in the problem. The implicit integration formula is simply one more equation to be converged through the normal iteration.

One other issue needs to be dealt with, and that is preserving past values of dynamic variables and their derivatives. From Equation 2.6 (See page 23) we see that the Euler integration formula uses values of *x* and $\dot{x}$ from the previous time to calculate *x* at the new time. Some integration formulas use values of these quantities from earlier time steps as well. In order to provide these past values, SPARK provides four past values for all problem variables. This allows definition of a wide range of practical integrator classes.[10]

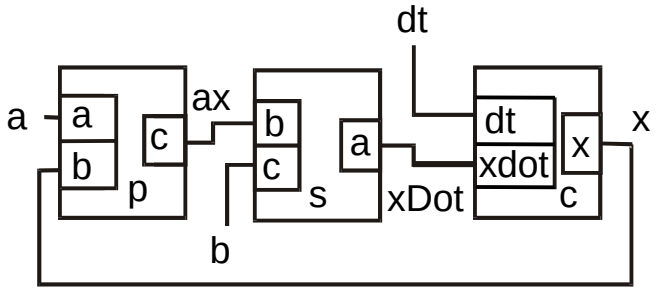

Figure 2.6 First order differential equation diagram.

With these ideas we can continue with our example. Figure 2.6 shows a SPARK diagram for our differential equation. We use an instance of the **safprod** object class, **p**, to form the ax product, and an instance of the **sum** object class, **s**, to form the sum $\dot{x} + ax$. We then link the **a** *port* of **s** and the $\dot{x}$ *port* of the **Euler** object, ***c***, using a problem variable called **xDot**. This causes the *x port* of *c* to carry the problem variable, *x*, which we also link to one of the multiplicand ports of **p**.

[10] If needed, SPARK can be reconfigured to allow more past values.

```
/*        First order differential  equation
 *                xdot + a*x = b
 *               frst_ord.pr
 */
declare     safprod    p;
declare     sum        s;
declare     euler      c;
input  a     p.a;
input  b     s.c;
link   dt    c.dt             global_time_step;
link   x     p.b, c.x         report;
link   xDot  s.a, c.xdot;
link   ax    s.b, p.c;
```

The values of **a** and **b**, must be placed in an input file, *frst_ord.inp*. Also, when you solve differential equations it is necessary to provide initial conditions for each dynamic variable. In SPARK there are two ways to accomplish this. One way is to place *init=value* in the *link* statement for the variable. Alternatively, you can specify the initial values by giving the initial time and associated initial values for the dynamic variables in the input file. This is preferable if you want to carry out parametric runs with different initial conditions without changing the problem specification file. To demonstrate the latter method, suppose **a** and **b** are 1.0 and 1.0, respectively, the initial time is to be 0, and *x* is to have an initial value of 0. Then *frst_ord.inp* should be:

```
3          a          b          x
0          1.0        1.0        0.0
```

Since *x* is a dynamic variable rather than specified as *input*, its value will be read from the input file *only at start-up*. In some numerical integration methods require values of dynamic variables and their derivatives at times earlier than the initial time. When needed, these values can be provided in the same manner, using time values earlier than the problem initial time (i.e., negative time if initial time is 0).

Note that the units of time are not defined in SPARK, so you are free to choose whatever time units you wish. You simple develop your differential equations to reflect your choice. For example, if in the above differential equations *x* is measured in meters and $\dot{x}$ is to be in meters/second, the coefficient *a* must have units of reciprocal seconds and *b* must have units of meters/second.

The run control file needed to run this problem, *frst_ord.run*, is:

```
(
InitialTime       ( 0.0 ())
FinalTime         ( 5.0 ())
TimeIncrement     ( 0.015625 ())
FirstReport       ( 0.0 ())
ReportCycle       ( 0.03125 ())
InputFiles        ( frst_ord.inp ())
OutputFile        ( frst_ord.out ())
)
```

We ask for the solution over a time range of 0 to 5 seconds, with a time step of 0.015625.[11] The *link* for *dt* includes the *global_time_step* keyword. This propagates the time step specified in the run control file to wherever it may be needed in the problem and macro classes. The requested output at every other time step is written to *frst_ord.out*. The results are plotted in Figure 2.7, generated by opening *frst_ord.out* with

[11] Although the time step can be any wanted value we choose $1/2^6$ =0.015625 because powers of 2 can be represented exactly in the binary storage format used internally. Step sizes that are not powers of 2 are difficult to synchronize with reporting intervals.

Microsoft Excel. Alternatively, you could use the free-use plotting program provided with WinSPARK, *wgnuplot*. (See Section. 3.12, page 63).[12]

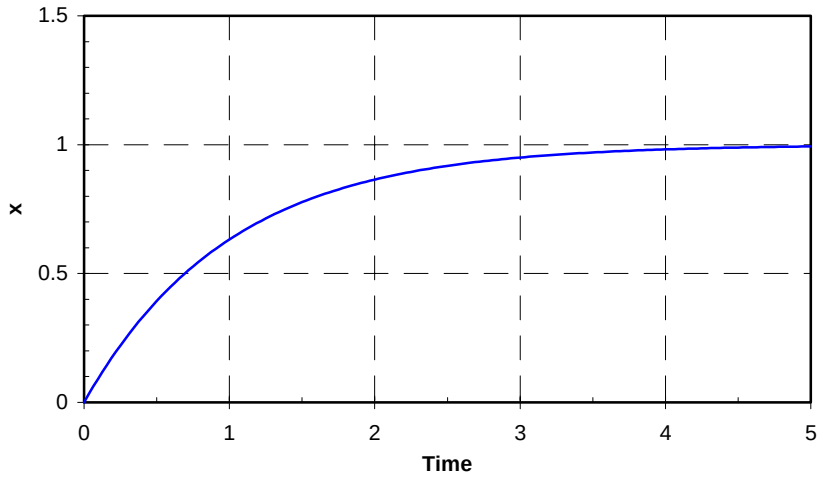

Figure 2.7 Results for frst_ord problem.

2.6.4 SPARK Library Integrator Object Classes

The SPARK library has several integrator object classes. These are shown in Table 2.2. All of these methods are fully described in numerical analysis texts so we will just describe them briefly here.

Table 2.2 Integrator Object Classes in the SPARK Library.

Method	**Class file**
Euler integrator (explicit)	euler.cm
Implicit Euler integration	implicit_euler.cc
Backward-forward difference	bfd.cc
4th-order backward - forward difference	bd4.cc
Adams-Bashforth-Moulton	abm.cc

The Euler object is based on the simplest of all methods, using only the derivative at the beginning of the time step. The implicit Euler method is the same basic idea as the normal (explicit) Euler method except the derivative is estimated as the average of that at the beginning and that at the end of the time step. The backward-forward difference method is only slightly more complex, using the derivative at the end of the time step as well as at the beginning. The 4^{th}-order backward-forward difference method uses additional previous values and derivatives. These bfd methods are often used for "stiff" differential equations sets (Press, Flannery et al. 1988).

[12] Although not included in the *VisualSPARK* release, *gnuplot* is available at various Internet sites and will run on UNIX as well as Windows platforms.

The Adams-Bashforth-Moulton method is a predictor/corrector method. Such methods employ two separate integration formulas, a *predictor* to make an initial estimate of the new solution, and a *corrector* to refine the solution iteratively. Naturally, the predictor is an explicit formula, while the corrector is implicit.

2.6.5 Creating SPARK Integrator Object Classes

If none of the library integrator object classes are suitable, you can define your own. SPARK integrator object classes are created much like any other object class. To see how this is done, let's look at the definition of the **Euler** class. The port variables are the dynamic variable **x**, its derivative **xdot**, and the time step **dt**. An inverse is given for a single port variable, the dynamic variable **x**.[13]

The port variables are the dynamic variable **x**, its derivative **xdot**, the time step **dt**. An inverse is given for a single port variable, the dynamic variable **x**.

```
/*          euler.cc             */
#ifdef spark_parser
port  x;
port  xdot;
port  dt;
functions {
      x = euler(x,xdot,dt);
}
equations {
      x = x_p + dt*xdot_p;
}
#endif      /*spark_parser*/
#include "spark.h"
double euler(ArgList args) {
      const double x_p = args[0][1];    // previous x
      const double xdot_p = args[1][1]; // previous xdot
      const double dt = args[2];        // time step
      if (::IsInitialTime())
            return args[0].GetInit();
      else
            return x_p + dt*xdot_p;
}
```

The function *Euler* employed in the class definition is expressed in *C++* after the class itself. It is basically an expression of the Euler integration formula, Equation 2.6. As might be surmised from the code, *args[i][1]* refers to the value of the i^{th} argument one time step back. Since the first argument is *x* and the second is *xdot*, *args[0][1]* and *args[1][1]* are the previous values of *x* and *xdot*, respectively. Initializing *const doubles, x_p, xdot_p* and *dt*, to corresponding elements of the *args* array is equivalent to *#define* used in earlier examples (See Section 2.4.2, page 16).

The heart of the function is the line:

```
      return x_p + dt*xdot_p;
```

which represents the Euler formula. The right hand side adds the time step multiplied by the derivative at the beginning of the time step to the variable at the same time. This is the new value of the dynamic variable, which is then returned. This gets executed at every solution time except the first. When time is

[13] Theoretically, SPARK would not care whether the integration formula was used to calculate the dynamic variable or its derivative. As a token to the sensibilities of most numerical analysts, however, here we restrict this relationship to be a formula for the dynamic variable.

InitialTime, the function returns the user-specified initial value of the dynamic variable (See Section 3.1, page 37).

More complex integrators, differing primarily in the use of more previous terms, may be found in the SPARK *globalclass* directory. There it will be seen that *args[i]* two steps back is written *x[i][2]*, and so on. Users with special needs can reconfigure SPARK to work with any number of previous values of any class argument.

In addition to using a single previous value, the integrator in this example is also simplified in another way. As presented, it uses the same variable name, *x*, to represent both the new value to be computed at the time step and the previous value. That is, **euler.cc** has the line:

```
x = euler(x, xdot, dt);
```

where *x* appears on both sides. Written this way, the SPARK parser will assume that we are using the current-time value of *x* in the right hand side of the integration formula, whereas in fact it is the *previous*-time value of *x* that occurs there (See Equation 2.9, page 24). Since the code for the corresponding *C++* function **euler()** actually uses only the previous value of *x* on the right hand side, namely *args[0][1]*, *Euler* integration will be properly applied at execution time. However, the disadvantage of the way we have coded it here is that the generated solver will include an unnecessary iteration loop. A better way to implement integrators is discussed in Section 3.9 (See page 51).

2.7 A Larger Example: Air-Conditioned Room

As a more realistic simulation example let us consider a simple air-conditioned room as shown in Figure 2.8.

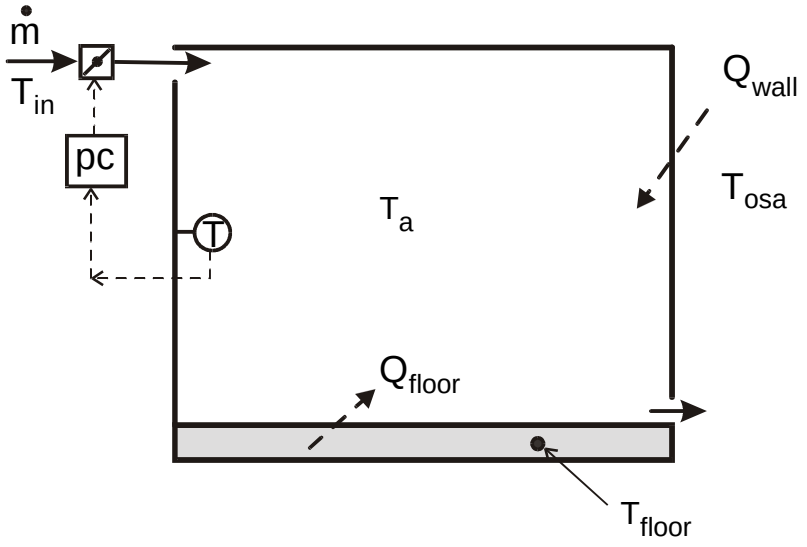

Figure 2.8 Temperature controlled room.

It is supplied by air at temperature T_{in}. The flow rate of supply air is $\dot{m}$, which is controlled by a proportional controller acting in response to the difference between room air temperature, T_a, and the set point, limited between maximum and minimum values T_{min} and T_{max}. Heat Q_{wall} is transferred through the external envelope in proportion to the outside-to-inside temperature difference. Also, heat Q_{floor} is transferred from the floor slab to the room air in proportion to the temperature difference between these two bodies. Accounting for the heat capacity of the floor slab, the mathematical model for this system can be written:

$$\begin{cases} Q_{wall} = UA_{wall} \cdot (T_a - T_{osa}) \\ Q_{floor} = hA_{floor} \cdot (T_a - T_{floor}) \\ Q_{flow} = \dot{m}Cp \cdot (T_{in} - T_a) \\ Q_{floor} = Q_{flow} - Q_{wall} \\ MCp_{floor} \cdot \dot{T}_{floor} = Q_{floor} \\ \dot{m}Cp = \max\left(\min\left((T_a - T_{\min}) \cdot \dfrac{(\dot{m}Cp_{\max} - \dot{m}Cp_{\min})}{(T_{\max} - T_{\min})}, \dot{m}Cp_{max} \right), \dot{m}Cp_{min} \right) \end{cases} \quad (2.11)$$

where:

UA_{wall} is the wall conductance,

T_{osa} is the outside air temperature,

hA_{floor} is the floor to room air conductance,

T_{floor} is the floor slab temperature,

T_a is the room air temperature,

Q_{wall} is the heat flow from room air to walls,

Q_{floor} is the heat flow from room air to floor,

Q_{flow} is the heat added (+) or removed (-) from the room due to air flow,

$\dot{m}Cp$ is the supply air capacity rate,

MCp_{floor} is the floor slab heat capacity,

$\dot{m}Cp_{\max}$ is the maximum supply air capacity rate,

$\dot{m}Cp_{\min}$ is the minimum supply air capacity rate,

T_{min} is the room temperature at which supply air capacity rate is maximum, and

T_{max} is the room temperature at which supply air capacity rate is minimum.

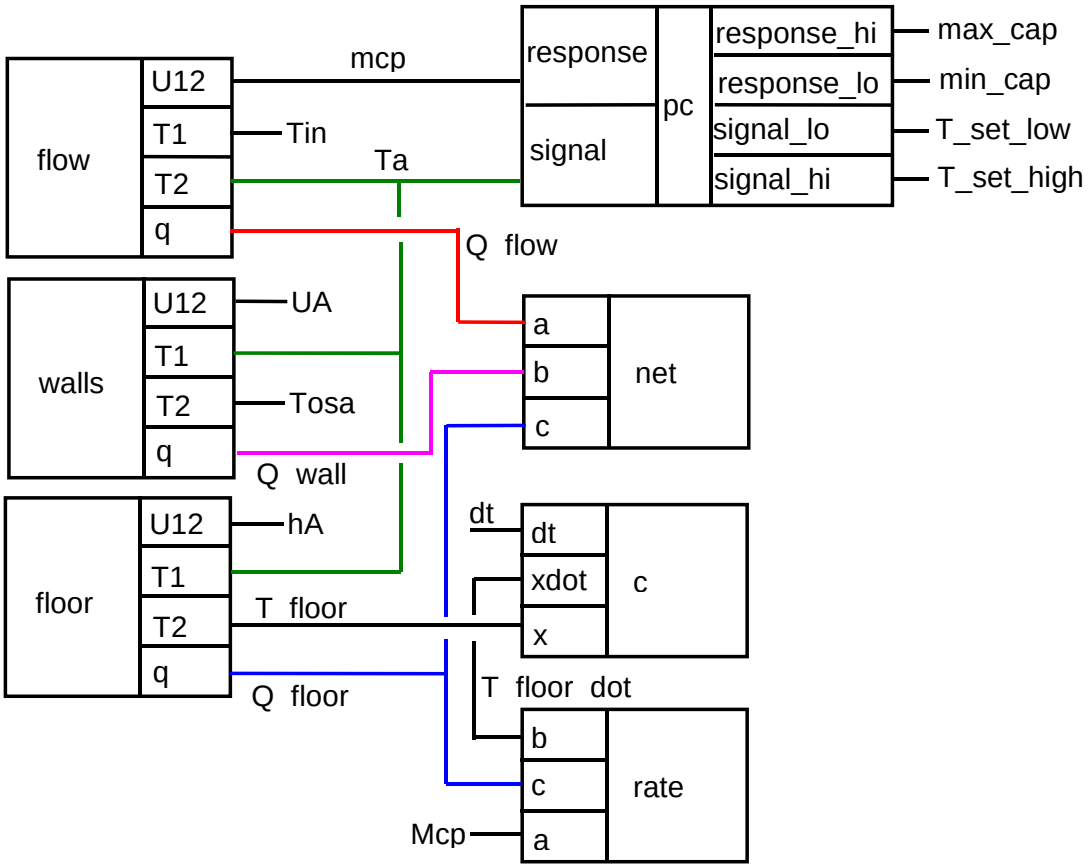

Figure 2.9 SPARK diagram for temperature controlled room (See macro class `room_fc.cm`*).*

The first two equations express the relationship between the temperature differences and heat flow to the room air, while the third gives the heat removal rate due to the stream of conditioned air. The next two give, respectively, the heat storage rate of the floor slab, Q_{floor}, and the rate of change of energy stored in the slab, $MCp_{floor} \cdot \dot{T}_{floor}$; of course, these quantities are equal. The last equation is the proportional control expression, stating that the air stream cooling capacity is proportional to the difference between room air temperature and the set point, limited between maximum and minimum values.

This system can be represented by seven SPARK objects as shown in Figure 2.9. The three heat transfer equations are represented by the objects *flow*, *walls*, and *floor*, all of which are instances of the HVAC tool kit class called **cond** (a conductor) having the form:

$$q = U12 \cdot (T1 - T2) \tag{2.12}$$

The slab heat storage rate relationship is represented by a **diff** object called *net*. Also, a **safprod** object called *rate* is required to form a product between the slab heat capacity, *MCp*, and the rate of change of slab temperature, *T_floor_dot*. An integrator object called *c* implements the backward-forward difference formula to get *T_floor* from *T_floor_dot*. Finally, the proportional controller is implemented by the class called *propcont* from the HVAC Toolkit (See Appendix A on page 81).

Because several rooms are often required in a complete problem, we implement the diagram in Figure 2.9 as a SPARK macro class called `room_fc.cm`, as shown below:

```
/*                Massive Floor Room, with Controller
 *                     Macro
 *                   room_fc.cm
 */
// Temperatures
PORT  Ta          [deg_C]   "Room air temperature";
PORT  T_floor     [deg_C]   "Room floor temperature";
PORT  T_floor_dot [deg_C/s] "Room floor temperature rate of change";
PORT  Tosa        [deg_C]   "Outside air temperature";
PORT  Tin         [deg_C]   "Supply air temperature";

PORT  UA          [W/deg_C] "Wall conductance";
PORT  hA          [W/deg_C] "Floor to air conductance";
PORT  mcp         [W/deg_C] "Supply air heat capacity rate";
PORT  Mcp         [J/deg_C] "Floor mass heat capacity";

// Proportional controller
PORT  T_set_high  [deg_C]   "Set point temp, high";
PORT  T_set_low   [deg_C]   "Set point temp, low";
PORT  max_cap     [W/deg_C] "Max supply air capacity rate";
PORT  min_cap     [W/deg_C] "Min supply air capacity rate";

// Heat transfers
PORT  Q_flow      [W]       "Heat added (+) /removed (-) by air stream";
PORT  Q_wall      [W]       "Wall heat transfer";
PORT  Q_floor     [W]       "Heat from air to floor";

PORT  dt          [s]       "Time step for T_floor differential";

DECLARE  cond     flow;   //  Air mass flow "conductor"
DECLARE  cond     walls;  //  Walls conductance
DECLARE  cond     floor;  //  Floor to air conductor
DECLARE  diff     net;    //  Diff between Q in and out
DECLARE  safprod  rate;   //  Multiply T_floor_dot* Mcp
DECLARE  propcont pc;     //  Proportional controller
DECLARE  bfd      c;      //  Backward-forward difference integrator

LINK  .Tosa,        walls.T2;
LINK  .Tin,         flow.T1;
LINK  .UA,          walls.U12;
LINK  .hA,          floor.U12;
LINK  .mcp,         flow.U12, pc.response;
LINK  .Mcp,         rate.a;
LINK  .T_set_low,   pc.signal_lo;
LINK  .T_set_high,  pc.signal_hi;
LINK  .max_cap,     pc.response_hi;
LINK  .min_cap,     pc.response_lo;
LINK  .Q_wall,      walls.q,  net.b;
LINK  .T_floor,     floor.T2, c.x;
```

```
LINK  .T_floor_dot, rate.b,   c.xdot;
LINK  .Q_floor,     floor.q,  net.c,    rate.c;
LINK  .Ta,          flow.T2,  walls.T1, floor.T1, pc.signal   INIT=20.0;
LINK  .Q_flow,      flow.q,   net.a;
LINK  .dt,          c.dt;
```

This macro can be used to define a single-room problem as follows:

```
/*                Air-conditioned Room
 *                  room_fc.pr
 */
DECLARE room_fc  room;

INPUT Mcp        room.Mcp         [J/deg_C];
INPUT UA         room.UA          [W/deg_c];
INPUT hA         room.hA          [W/deg_C];
INPUT Tosa       room.Tosa        [deg_C];
INPUT Tin        room.Tin         [deg_C];
INPUT T_set_high room.T_set_high  [deg_C];
INPUT T_set_low  room.T_set_low   [deg_C];
INPUT max_cap    room.max_cap     [W];
INPUT min_cap    room.min_cap     [W];

LINK dt          room.dt          [s]                   GLOBAL_TIME_STEP;
LINK mcp         room.mcp         [W/deg_C]                  REPORT;
LINK Q_flow      room.Q_flow      [W]                        REPORT;
LINK Q_wall      room.Q_wall      [W]                        REPORT;
LINK Q_floor     room.Q_floor     [W]                        REPORT;
LINK Ta          room.Ta          [deg_C] BREAK_LEVEL=10     REPORT;
LINK T_floor     room.T_floor     [deg_C] INIT=30            REPORT;
LINK T_floor_dot room.T_floor_dot [deg_C/s]                  REPORT;
```

Here we have declared *room* as an instance of the **room_fc** macro class. The room thermal characteristics and control settings are defined as *input*s. This alone would be sufficient to completely specify the problem since the necessary linkages are all internal to the **room_fc** macro class. However, if we did not put some *link* statements in the problem file, SPARK would have no problem variables and hence nothing to report. We therefore introduce *link* statements to get reports on the room air temperature, *Ta*, floor slab temperature, *T_floor*, cooling rate of the air stream, *Q_flow*, and the air stream capacity rate, *mcp*. Alternatively, one could use the *probe* keyword, Section 3.7 (See Page 49).

The input data for this problem is shown in Table 2.3 (See Page 36). Note that the supply air temperature is initially 13°C, and is raised to 17°C at 20 hours (72,000 seconds) after starting. The *room_fc.inp* file to specify this is constructed as shown below:

9	hA	UA	Tosa	Tin	Mcp	T_set_low	T_set_high	max_cap	min_cap
0	60	30	38	13	1.e6	23	24	50	0
71964	60	30	38	13	1.e6	23	24	50	0
72000	60	30	38	17	1.e6	23	24	50	0
*									

In the first line the first item, 9, is the number of problem input variables. The next nine items in this line are the names of the input variables as defined in the *input* statements in the problem specification file. The data that follow give the times (in this case, seconds) and values for the inputs at discrete points throughout the intended simulation period. The first line, with a time value of 0, gives the initial conditions. We specify T_{in} to be set at 13°C from time 0 to 19.99 hours (71,964 seconds), and 17°C from 20.0 hours (72,000 seconds) forward. Other values are constant throughout the simulation. SPARK will interpolate linearly between the given time values to arrive at the value of all input variables at each solution point as the simulation proceeds.[14] The last line has an asterisk, *, meaning that all values remain fixed from that point forward.

It will be observed that the time unit in the above example is seconds. While there is a certain awkwardness with this choice, it has the advantage of allowing the other problem variables to be expressed in true SI units. For example, had we chosen to use hours instead of seconds, the time values would be the (perhaps) more pleasing sequence 0, 19.99, 20.00, but then we would have had to express input data such as hA_f in J/(hour*deg_C) instead of W/deg_C.

Another observation in this example is that some input values do not vary with time, and this leads to many repeated values in the file. While there is nothing wrong with repeating the constant values as done here, there are alternatives that you may want to consider. Perhaps the best way to deal with this situation is with multiple input files, as discussed in Section 3.4 (See page 43). Another way to deal with a constant input variable, not necessarily recommended, is simply to omit it from the input file. This sometimes works because problem input variables not listed in an input file will assume their *init* values, if available. *Init* values are specified in the *port* statement (Section 4.9, page 74) when SPARK classes are defined. If the class does not provide *init* values, or the provided values are not acceptable, you can also give an *init* value on a *link* connected to the *port*. The disadvantage of doing it this way is that the problem must be rebuilt whenever *init* values are changed.

However provided, running the *room_fc* problem with the data in Table 2.3 produces the results plotted in Figure 2.10 and Figure 2.11 (See Page 34).[15]

All *input*s are constant except *Tin*, which starts at 13°C and is increased to 17°C at 20 hours (72,000 s). The first of these plots, Figure 2.10, shows the controlled quantity, *mcp*, and we see that it remains at its maximum value for about six hours. During this period the room air temperature, Figure 2.11, is being rapidly reduced. Once within the range of proportional control, the supply capacity rate modulates, maintaining the room air temperature close to the set point. The slab temperature gradually cools. At the twentieth hour, the scheduled change in supply air temperature takes place, causing the supply capacity rate to increase to the maximum. However, this maximum is insufficient so the air temperature rises above the set point.

[14] Note that there must be some time difference between successive points to allow legitimate interpolation.

[15] To get these plots we opened the output file with Microsoft Excel. Alternatively, *gnuplot* could be used.

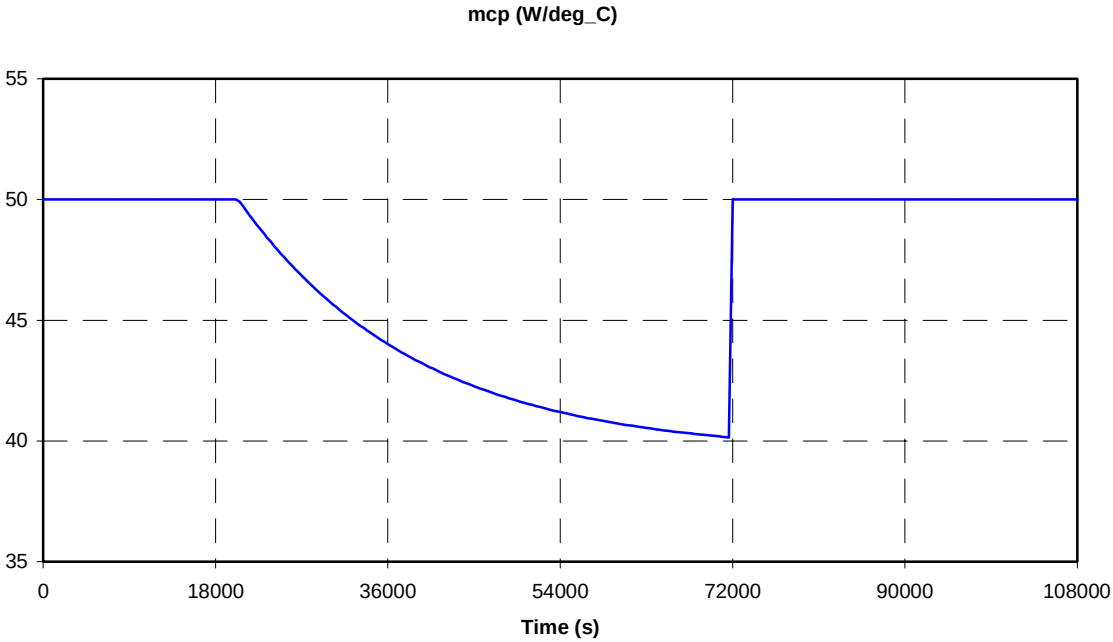

Figure 2.10 Supply Air Capacity Rate.

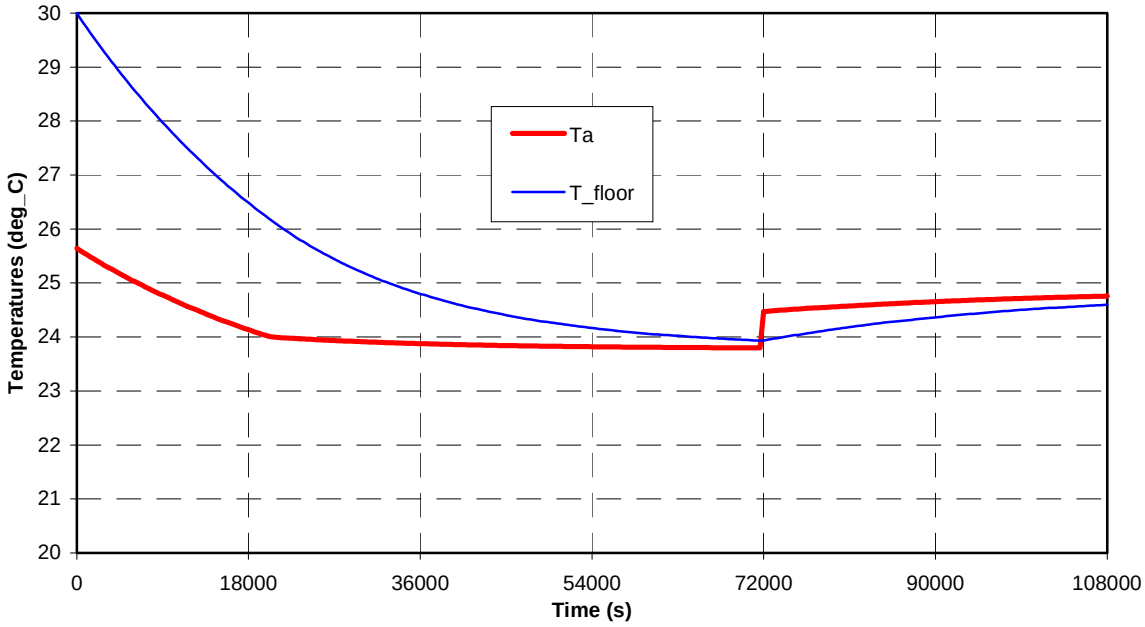

Figure 2.11 Room and Floor Slab Temperatures.

The *room_fc.eqs* file, below, reveals how SPARK solves this problem. We see that there is a single strongly connected component, with two break variables, *Ta* and *T_floor*. The initial values of *Ta* is taken from the *init* values found in the macro or underlying atomic classes, since it is not mentioned in the input file, and no *init* value is given in the *link* statement in the problem file. *T_floor* is initialized at *InitialTime* at the *init* value given in the *link* statement in the problem file. These plus the problem inputs allow the indicated sequence of calculations. The component is iterated to convergence at each time step.

```
DT:
        dt <- dt
Inputs:
        max_cap
        min_cap
        T_set_high
        T_set_low
        Mcp
        hA
        UA
        Tosa
        Tin

Component 0:
        mcp = propcont(Ta, T_set_low, T_set_high, min_cap, max_cap)
        Q_flow = cond_q(Tin, Ta, mcp)
        Q_wall = cond_q(Ta, Tosa, UA)
        Q_floor = diff_difference(Q_flow, Q_wall)
        T_floor_dot = safprod_quot(Q_floor, Mcp)
[break] T_floor = bfd(T_floor, T_floor_dot, dt)
            = T_floor [predictor]
[break] Ta = cond_T1(Q_floor, T_floor, hA)
            = Ta [predictor]
```

Table 2.3 Input for the Controlled Room Example.

Variable (See Equation 2.11)	**Link** (See `room_fc.pr`)	**Value**	**Units**
hA_{floor}	hA	60	W/deg_C
UA_{wall}	UA	30	W/deg_C
T_{osa}	Tosa	38	deg_C
$\dot{m}Cp_{\min}$	min_cap	0.0	W/deg_C
$\dot{m}Cp_{\max}$	max_cap	50	W/deg_C
T_{set_high}	T_set_high	24	deg_C
T_{set_low}	T_set_low	23	deg_C
dt	dt	360	s
MCp_{floor}	Mcp	1.0 e6	J/deg_C
T_{in}(0-71964)	Tin	13	deg_C
T_{in}(72000-...)	Tin	17	deg_C
T_{floor}(0)	T_floor	30	deg_C

Section 3 Advanced Topics

3.1 Numerical Integration Issues

As discussed in Section 2.6 (See page 22), solution of differential equations in SPARK requires using integrator objects in the problem description. We saw a simple integrator class in Section 2.6.5 (See page 27). Although this is basically a simple idea, there are some details, mostly having to do with start-up at the problem initial time, that advanced users need to be aware of.

To understand these issues, we need to consider the situation at the very beginning of the simulation period, i.e., *InitialTime*, and contrast it with conditions at later time steps. At *InitialTime*, presumably we want the prescribed initial values of the dynamic variables to used. That is, if *x* is a dynamic variable we want to enforce:

$$x = x(t_0) \tag{3.1}$$

where $x(t_0)$ is the prescribed initial value of *x*. However, at all other times in the solution we need to calculate the value of *x* from the integration formula used in the SPARK integrator object. That is, assuming we are using the Euler formula, we want to enforce:

$$x = x_p + h\dot{x}_p \tag{3.2}$$

where the subscript *p* refers to the previous time step values for *x* and its derivative. Thus we see that the system model is slightly different at *InitialTime*. Ideally, then, we should formulate the problem twice, once with an object representing Equation 3.1 and again with the integrator relationship, Equation 3.2, starting the simulation with the first formulation and switching to the second after the *InitialTime* solution. However, SPARK 1.0 can not change the model during simulation; it allows for a single problem formulation. Therefore we have to use the integrator object, Equation 3.2, at *InitialTime* as well as throughout the simulation period.

There are two options you can use in SPARK 1.0 to achieve proper start-up. One approach is to observe that the difficulty arises because there are no proper "previous" values for *x* at *InitialTime*. However, we can turn this to our advantage by simply assigning values of x_p and $\dot{x}_p$ that will result in the Equation 3.2 object producing the same value for *x* as Equation 3.1, i.e., $x = x(t_0)$. For example, setting both x_p and $\dot{x}_p$ to 0 will work if the initial value of *x* is supposed to be 0. In Section 3.4 (See page 43) we see how this can be done in input files. There are disadvantages to this approach. First, determining values for x_p and $\dot{x}_p$ that will produce the wanted initial values of *x* is awkward even for simple problems. The method

becomes intractable when more complex integration formulas are used, especially if more than one previous value is employed.

Another approach that is often more attractive is to modify the integrator class to use a different inverse function at Initial Time. For example, we could write

```
if (::IsInitialTime())
      return args[0].GetInit();
else
      return x_p + h*xDot_p
```

where *IsInitialTime()* is a boolean function (see Section 3.15, page 67) that returns True only when time equals *InitialTime*, and a*rgs[0].GetInit()* is a function that returns the initial value of a*rgs[0]*. This is actually quite a good solution to the start-up problem. It is easy to implement and will adapt to even complex integrators. The drawbacks are small losses in computational efficiency and generality. The principal efficiency loss is due to the extra *if-check* which must be executed at every time step in the simulation; it is doubtful that this increase in solution time will be significant in most problems. The loss in generality is because certain kinds of initial conditions, e.g., $\dot{x}(t_0) = c$, can not be enforced. SPARK 2.0 will deal with this start-up situation more rigorously. Two different problem graphs will be constructed, one using a start-up formula and the other a proper integration formula. This will allow determination of completely different solution sequences at start up if needed to enforce special initial conditions. Moreover, this approach will permit use of different integration formulas whenever necessary later in the simulation, e.g., after a change in integration step size.

3.2 Iterative Solution and Break Variables

As we have noted in earlier examples, systems of equations often have to be solved iteratively. In SPARK, this can be true even if the equations are all linear, because no specific test is done for linearity. Normally, the user need not be concerned with the iterative process, so we will not go into detail here. However, a general awareness of the methods used is helpful if solution difficulties are encountered.

First, in the problem setup phase, SPARK determines if iteration is required by detecting cycles in the problem graph. If cycles are detected, a graph algorithm is used to find a small set of variables (nodes in the graph) that "cut" the cycles. The associated problem variables, called "break variables," are placed in a vector to act as the unknown vector $\bar{x}$ in a multi-dimensional Newton-Raphson solution scheme. The functions that are forced to zero in the Newton-Raphson process are of the form

$$g(x) = f(x) - x \tag{3.3}$$

where x is the vector of break variables, and $f(x)$ represents the directed acyclic graph formed when the original problem graph is cut at the cut-set vertices. In other words, the current solution estimate, *x*, is applied to the graph, producing $f(x)$, from which the original estimate is subtracted. At the solution, $f(x) - x = 0$. The Jacobian matrix for the Newton-Raphson process is then $J = \frac{\partial g}{\partial x}$. In each Newton-Raphson iteration the next estimate is calculated by solving the linear set

$$J\Delta x = g(x) \tag{3.4}$$

for Δx, then calculating

$$x^{k+1} = x^k - \Delta x \tag{3.5}$$

The solution of the linear set, Equation 3.4, is carried out with Gaussian elimination, LU decomposition, or similar method. Note that the size of *J* is the size of the cut set, so this solution can be much more efficient than if we had not attempted to minimize the cut set.

Normally, this process converges to the solution quite rapidly (quadratically). However, it is well known that the Newton-Raphson process, like all methods for solving general sets of nonlinear equations, can fail to converge under certain circumstances. Failure occurs when the residual functions have particular kinds of nonlinearities and the starting values are not sufficiently close to the actual solution. Thus starting values are important.

In SPARK, we refer to the process of selecting a starting value for the iteration process as "prediction". By default, the prediction for solution at a particular time step is the final solution value for the same variable at the previous time step. This can be changed by use of the *pred_from_link=linkFrom* keyword in the corresponding *link* statement. In this case, the value of the *linkFrom link* is used as the predictor. Note that at the *InitialTime* solution there is no proper "previous time step value." In this case, if there is no *pred_from_link=linkFrom,* SPARK will use the default value for the break variable as the initial predictor. Since default values determined in this way are not appropriate for every variable, they may not be very close to the solution value. Therefore it is best to provide *initial predictor*s via input files. This issue is discussed further in Sections 3.3.1 (See page 39) and 3.3.2 (See page 40).

3.3 How SPARK Assigns Values to Variables

In a broad sense, one would think that variable values in a problem should either be specified by the user, or be calculated in the process of solving the problem. While this is indeed true, there are issues having to do with SPARK value assignments that sometimes need careful attention. This is best discussed in terms of four different methods of value assignment that can take place in SPARK: initialization, prediction, updating, and solution.

3.3.1 Initialization

Initialization refers to providing values that are needed at the beginning of the simulation. Using these initial values, SPARK then computes values for all link variables at the initial time of the simulation. While all SPARK variables can be initialized, not all need to be initialized.

What Must be Initialized

There are two cases where variables must be given initial values, regardless of the numerical methods to be used:

Dynamic variables. These are the *link* variables that appear in differential equations, i.e., those attached to an *x* port of integrators. This initialization requirement arises directly from the underlying mathematical theory, namely that you need an initial condition, in addition to the differential equation, in order to have a well posed problem. This requirement is independent of the choice of integration method or other numerical considerations.

FromLinks of Previous Value Variables. Previous Value Variables (See Section 3.9, page 51) are in a special category in SPARK. Most SPARK non-input *link* variables get values in the process of solving the problem equations at the time in question. Previous Value Variables, on the other hand, get their values *from calculations done at the previous time step*. As described in Section 4.13 (See page 77), the syntax *update_from_link = fromLink* defines the *link* from which the variable in question gets its value. For this to work properly at *InitialTime*, obviously the variable referred to as *fromLink* must be initialized at the time *one time step before the problem initial time.* This can be done either in an input file, or using the *init* in the

link statement defining *fromLink*. Note, however, that Previous Value Variables that arise in the definition of integrators need not be initialized because they are never used at *InitialTime*.

What Might Need Initialization

Additionally, certain numerical integration methods may need to be initialized not only at *InitialTime*, but also at one or more earlier time steps. While this can be done in SPARK, as a practical matter it is difficult or impossible to know such values. For this reason, SPARK 2 will avoid use of such methods at the beginning of the simulation and until necessary histories of past values have been solved for with single step methods. For Spark 1.0, ideally you should attempt to provide past values as needed by multistep methods, if used. That said, some analysts may be willing to accept some degree of inaccuracy in early time steps, in which case this advice can be disregarded.

BDF-like multistep schemes require past values for the dynamic variables, as many as the order of the method. For example, the *bd4.cc* class requires values at one, two, three, and four time steps before the initial time of the simulation. Similarly, Adams-like multistep schemes, e.g., *bfd.cc* and *abm.cc*, require past values for the derivatives of the dynamic variables, again as many as the order of the method.

Finally, it should be noted that variables that SPARK selects as break variables may need initialization. The reason for this is that unless the *link* statement for break variable has the keyword *pred_from_link = fromLink* (see below) the iteration process at each new time begins at the previous value of the break variable. Without proper initialization, the previous value at *InitialTime* would likely become the built-in SPARK default value, 0.01. To override use of the default value, you must initialize the break variable at *InitialTime*.

How to Specify Initialization

The user can specify initial values in two ways. First, *init = value* can be placed in the *link* statement for the variable, or in any equivalent *link* to a *port* statement in macro objects (See Section 3.3.5, page 42). An alternative way to initialize is by means of input files. During the initialization phase of the simulation, all variables can have initial and past values assigned through reading from input files. This is done by providing the required variables and derivatives with values at *InitialTime*, and earlier time steps if needed, using negative times if necessary.

Initialization of Previous Value Variables is a special situation. Since a variable of this kind gets its value from the previous value of another variable, the proper way to provide its *InitialTime* value is to specify the value of the corresponding *fromLink* at one time step before *InitialTime*, indicated by a initial time minus the time step, using negative time stamp in an input file. Note that an attempt to use the *init* keyword in a *link* statement in which the *update_from_link* keyword is used results in a warning. Moreover, values given for Previous Value Variables *per se* in input files will be ignored.

Thus we see that SPARK initial values can come from the default values, *init = value*, or input files. Either of the latter two will override the first. If a variable has both *init = value* and occurs in an input file, the file input overrides the *init* value.

3.3.2 Prediction

In SPARK *prediction* refers to providing values for *break variables* at the beginning of each time step, i.e. prior to solving the simultaneous algebraic problem by iteration.

Where Prediction is Needed

As a rule, only break variables need predicted values.

How Prediction is Specified

By default, predicted values for break variables come from the final value for the same variable found at the previous time step. In many cases this will work well, so no special steps have to be taken by the user. If your problem encounters solution difficulties, you may want to provide better prediction using either the *pred_from_link* feature for *links*, or the *pred* feature in the class definition.

If *pred_from_link = fromLink* appears in the *link* statement for a break variable, the starting value for the iterative solution at the new time will be the value of *fromLink*. This mechanism is used when you know that the value of *fromLink* provides a more reliable estimate for the break variable than its previous value. Note that since the *fromLink* can be any link, this mechanism allows you to devise predictor using variables from anywhere in your problem. Therefore it is a very general and powerful mechanism.

Another mechanism for prediction is provided by the syntax:

```
PRED = predictor_fun(port1, port2, port3, ...)
```

in the Functions section of a SPARK class definition. This methods provides a predictor at the *class level*, as opposed to the *pred_from_link* keyword which provides prediction at the *link level*. Class level prediction is primarily used to implement predictor-corrector integration schemes (e.g., *abm.cc*), where the predictor scheme is specified following the *pred* keyword. Another possible usage of class level prediction is to provide a predictor function for a nonlinear atomic class using a linearized form of the nonlinear equation. This approach has been successfully applied with the airflow-pressure power law relation in the zonal model context. Unlike link level predictors, class level predictors can involve only the variables connected to the ports of the class in question.

If a variable has both link and class level prediction (an unlikely situation), the class level prediction will override the link level prediction.

3.3.3 Updating

The concept of Previous Value Variables (See Section 3.9, page 51), requires the concept of *updating* as a means of assignment of values to such variables.

What Needs to Be Updated

Updating refers only to providing values for Previous Value Variables at the beginning of each time steps.

How Updating is Specified

To implement this concept, every Previous Value Variable has in its defining *link* statement:

```
update_from_link = fromLink
```

Previous Value Variables are viewed as receiving values by updating from the specified links. At the *beginning* of every time step, before solving the problem equations, the saved previous value of *fromLink* is assigned to the variable named in the link statement.

3.3.4 Solution

Solution is the prevalent method whereby values are assigned to variables in a SPARK problem.

What Needs to Be Solved For

Normally, values for SPARK variables are determined by the solution of the system equations at each time point in the solution interval. The exceptions to this are, Input variables, Previous Value Variables, and Dynamic variables at *InitialTime*

How Solution Is Specified

As noted earlier, keywords in the associated *link* statements often determine the role of the variable. Inputs variables are identified by the keyword *input* either replacing the *link* keyword, or occurring elsewhere in the *link* statement. Previous Value Variables are defined by the keyword *update_from_link* in the *link* statement. Dynamic variables, on the other hand, have no special identifying keyword. Variables become dynamic merely by being connected to an *x* port of an integrator. The absence of these special keywords in a link statement indicates that the associated variable is to be solved for.

Break variables are normal SPARK variables, other than inputs or Previous Value Variables, that happen to be selected by SPARK for iteration. Although they are assigned predicted values at the beginning of iteration at each time step, their final values after convergence at each time step are "solution" values, i.e., they satisfy the system equations. Note that the break variables are determined automatically by SPARK.

3.3.5 Propagation

As discussed previously, SPARK problem variables can have a default value assigned through the use of keywords in the *port* statement. This default value will replace the built-in default value (0.01) for the port. However, when SPARK atomic classes are used to build macro classes, and when both become parts of SPARK problem files, a question arises about precedence among these values as set at different levels. For example, suppose we define atomic class **ac1** which has a *port* called **T** with a *default* value of 20. Now suppose we define a macro class **mc1** which uses **ac1**, and this class also has a *port* called **T** with a *default* value of 10 which is linked to the **T** *port* of **ac1**. The question is, which default value will SPARK use for variables linked to the **mc1 T** *port* when it is used in a problem or another macro class? The same question can be posed for the *init*, *min*, and *max* values assigned through the *port* or *link* statements.

These questions are answered by propagation rules built into the SPARK parser. The first rule is that the *higher level takes precedence*. This means that a *default*, *init*, *min*, and *max* values given at any level override those given in lower level ports to which there is a connecting path. That is, values will automatically propagate downward as needed. Thus if *mc1* were to be used in a problem file (or another macro class), any variable linked to its T *port* would have a *default* value of 10.

Let us consider another facet of this problem. Suppose a *default* value is *not* given for the **T** *port* of the **mc1** discussed above. Will a variable linked to the **mc1 T** *port* have a *default* value (other than the built-in value of 0.01) when it is used in a problem or another macro class? The rule given above addresses *downward* propagation, but this question is one of *upward* flow of information, from a *port* in a low level class to a *port* linked to it in the higher level class or problem. To deal with this situation, SPARK applies a second propagation rule, which is that *default*, *init*, *min*, and *max* values are propagated upward through connected ports whenever the higher level ports have no corresponding values.

Together, these propagation rules produce behavior that most users will find natural. However, ambiguity can arise when a macro class *port* is linked to *two or more ports of constituent classes.* For example, suppose **mc1** also uses another atomic class, **ac2**, which also has a *port* called **T**, but with a *default* value of 15. Will the value propagated upward (in the absence of *default* specification of the **T** *port* in **mc1**) be 20 or 15? There is no way for SPARK to resolve such an ambiguity. Consequently, the propagated value will be determined by the order in which the parser encounters the linkages in **mc1**. To avoid such ambiguity, the user should assign values at the higher levels when building complex macro classes.

3.4 Input Values from Files

Most SPARK problems require data beyond that which is specified in the problem specification file. In particular, as we saw in the examples of Section 2, variables designated as *input* in the problem specification file need run time values. Moreover, certain other kinds of data are needed to specify exactly how the problem is to be solved numerically, e.g., initial values for dynamic variables and prediction values for iteration variables. All such data can be provided in SPARK input files. Although usually bearing the *.inp* extension, files of any extension can be used as SPARK input files..

Although in simple examples we have dealt with in this manual so far we have used a single input file for a SPARK problem, in practice it is often better to segregate the different kinds of input into separate files. One useful categorization of different types of input is:

Constant data: These are usually physical characteristics of the system that do not change with time. For example, surface areas, equipment capacities, and any other physical problem data that are assumed to be constant, such as heat transfer coefficients.

Time varying data: This includes any problem input data that varies with time during the simulation interval. The most common example in HVAC problems is weather data, such as ambient temperature and humidity. However, system control information, such as thermostatic set points, that are scheduled to change at particular times are also time varying inputs.

Initial Conditions: If the problem includes differential equations, the initial values of all dynamic variables must be provided. Although these can be specified in the problem specification file with the *init* keyword, it is usually better practice to specify them in an input file so they can be changed in subsequent runs without rebuilding the problem.

Numerical support data: Numerical techniques used in SPARK sometimes need, or at least benefit from, additional user supplied data. This category often includes initial predicted values for variables that are solved for by iteration, i.e., break variables. Also, if the chosen numerical integration methods for differential equations in the problem require previous values of the dynamic variables and/or their derivatives, they belong in this category.

In a well organized problem, each of these categories should have a separate input file. Moreover, it is sometimes wise to have multiple files within these categories. For example, you could have a separate constant data file for each subsystem in a complex model. Another situation calling for multiple input files within a category is when time varying data has different temporal characteristics. For example, if we wanted to have outside temperature T_{osa} varying hourly in the *room_fc* example it would be far easier to place this in a different file than the one with T_{in} which changes only once.

We can demonstrate these ideas by revisiting the **room_fc** example from Section 2.7 (See page 28). For example, we could create four separate input files using the above categories. The constant data file, appropriately called *room_fcDesignParameters.inp*, would contain:

```
8    hA    UA    Tosa    Mcp    T_set_low    T_set_high    max_cap    min_cap
0    60    30    38      1.e6   23           24            50         0
```

while the time varying data file, that we might call *room_fcTimeVaryingParameters.inp*, would contain:

```
1          Tin
0          13
71964      13
72000      17
*
```

Since the controlled room problem includes a differential equation, it is necessary to specify the initial value of the dynamic variable, *T_floor*. Rather than relying upon the *init* keyword to set the initial value for this

dynamic variable we can specify it in an initial conditions input file. This file could be called *room_fcInitialConditions.inp* and would contain:

```
1    T_floor
0    30
```

One advantage of this approach is that it is not necessary to rebuild the problem when initial values change.

Finally, we should create an input file for whatever information is needed to support the numerical solution process, provided such information is available. One issue in this regard is initial predictions for break variables, as explained in Section 3.3.2 (See page 40). As explained there, at the very beginning of the solution an initial predictor is needed because otherwise there would be no "previous time value" to use. If a reasonable estimate for a break variable is not readily available, SPARK can sometimes find a solution beginning with the default initial value, 0.01. However, if you can estimate more appropriate initial predictions the iteration process will have a better chance of quickly finding the correct solution at the start of the problem. Note that while better accuracy of these initial predictors will improve the chances for solution, usually great accuracy is not necessary.

In the case of the controlled room example the equation file reveals that SPARK chooses *Ta* and *T_floor* as break variables. There is no need to worry about *T_floor* in this regard, because the initial value already provided will be used directly. For the *Ta* variable, we can easily provide an estimate more accurate than the default value. For example, a value half way between the initial *T_floor* value and the supply air temperature value should be a reasonable for *Ta*. Thus a numerical support input file called *room_fcNumericalSupport.inp* could therefore be created as:

```
1    Ta
0    21.5
```

A problem run control file (See Section 3.14, page 65) must list the names and locations of all input files. For this example, we have *room_fc.run* as:

```
(
InitialTime       ( 0.0 ())
FinalTime         ( 108000.0 ())
TimeIncrement     ( 180 ())
FirstReport       ( 0.0 ())
ReportCycle       ( 360.0 ())
InputFiles        ( room_fcDesignParameters.inp ()
                    room_fcTimeVaryingParameters.inp ()
                    room_fcInitialConditions.inp ()
                    room_fcNumericalSupport.inp ()
                  )
OutputFile        ( room_fc.out ())
)
```

3.5 Macro Links

When systems with fluid flow are modeled, the component models are often connected with a common set of links. For example, HVAC system air components such as fans, heating and cooling coils, and mixing boxes are connected by links representing air enthalpy (or temperature), humidity, and mass flow rate.

In SPARK, a set of ordinary links such as these can be grouped together and used as a *macro link*, connecting *macro ports* of classes, thereby simplifying specification of such models.[16]

[16] Technically, a macro link does not exist in its own right as a SPARK construct. It is just a term for referring to a link connected to a *macro port*.

As an example of macro links and ports, consider a moist air mixer in which we define the interface to have three *macro port*s, representing two inlet flow streams and one outlet flow stream:

```
port AirEnt1      "Inlet air stream 1"    [airflow]
      , .m        "air mass flow"         [kg_dryAir/s]
      , .w        "hum. ratio"            [kg_water/kg_dryAir]
      , .h        "enthalpy"   NOERR      [J/kg_dryAir]
      ;
port AirEnt2      "Inlet air stream 2"    [airflow]
      , .m        "air mass flow"         [kg_dryAir/s]
      , .w        "hum. ratio"            [kg_water/kg_dryAir]
      , .h        "enthalpy"   NOERR      [J/kg_dryAir]
      ;
port AirLvg "Leaving air stream"          [airflow]
      , .m        "air mass flow"         [kg_dryAir/s]
      , .w        "hum. ratio"            [kg_water/kg_dryAir]
      , .h        "enthalpy"   NOERR      [J/kg_dryAir]
      ;
```

In this example, each *macro port* has three properties or subports, namely mass flow rate, humidity ratio, and enthalpy. Although the individual subports of one of these ports have separate names, description strings, and physical units, the *macro port* itself also has a name, description, and units string.[17]

When an object of this class is instantiated you can connect similar *macro port*s (i.e., those with like units and similar internal structure) in the same manner as you would connect ordinary ports. Thus if the class with the above interface were called **mixerMP** we could write (in some macro class or problem we were creating):

```
declare mixerMP m1, m2;
link AirStream1 m1.AirLvg, m2.AirEnt1;
```

This would connect the humidity ratio, mass flow rate, and enthalpy of the air stream leaving **m1** with the first inlet of **m2**.

Developing classes that use macro ports requires great care, since if it is not done correctly the objects will not connect properly. The principal requirement is that if the *macro port*s of two objects are to connect properly, the ports must be similarly defined in both objects. By "similarly defined," we mean that the unit strings for both *macro port*s must be identical, and that there must be at least one common port name between the two ports. This is no problem in the above example, since **m1** and **m2** are of the same class, and the leaving air *port* is defined exactly the same as the two entering ports.

However, errors can easily occur if the two ports being connected belong to objects of differing class, perhaps developed by different people. For example, suppose a fan class were to be defined with the entering air *port* defined as:

```
port AirEnt       "Inlet air stream"      [airflow]
      , .massFlow "air mass flow"         [kg_dryAir/s]
      , .w        "hum. ratio"            [kg_water/kg_dryAir]
      , .h        "enthalpy"   NOERR      [J/kg_dryAir]
      ;
```

Since the units string , **airflow**, is the same, SPARK would allow the following connection to be attempted:

```
declare mixerMP m1;
declare mfan f1;
link  InFlow m1.AirLvg, f1.AirEnt;
```

[17] Although, rather than physical units, the macro port "units" are merely a unique name, selected by the user.

However, since the flow subport is called **m** in the **mixerMP** and **massFlow** in the **mfan**, only the **w** and **h** subports would be successfully connected. This is because when the SPARK parser expands the macro link/port, it attempts to match subports of like names. If there are no subports in the second object that match any of the subports of the first, the parser rejects the *link* statement as erroneous. But if at least one of the subports at one end matches a subport at the other end, SPARK assumes you know what you are doing and accepts the link. This is useful since you may indeed want to connect some but not all subports; for example, you may wish to connect one component with a dry-air *macro port* (i.e., no humidity ratio) with another component that was designed for moist air calculations.[18]

There are also situations where you need to qualify individual subport in a macro link with one or more keywords. For example, suppose the first inlet *port* of **m1** in our first example comes from problem input data, and the mass flow rate is to be reported. The syntax to accomplish this is shown below:

```
declare mixerMP m1, m2;
link AirStream1   m1.AirLvg, m2.AirEnt1;
input massFlow1   m1.AirEnt1.m report;
input hFlow1      m1.AirEnt1.h;
input wFlow1      m1.AirEnt1.w;
```

As is seen in this example, this syntax is much the same as for ordinary links or inputs; the only difference is that we qualify the port name, e.g., **m**, with the subport name as a prefix. The dot (.) is used as a separator.

While the above syntax is valid and easy to interpret, it is not concise. A more concise syntax that expresses the same connections is:

```
declare mixerMP m1, m2;
link AirStream1 m1.AirEnt1 (.h) input (.w) input (.m) {input report};
link AirStream2 m1.AirLvg, m2.AirEnt1;
```

The first link statement defines a macro link called **AirSteam1** that is connected to the **AirEnt1** macro port of the **m1** object. We see that each subport is referenced with the notation (.portName), and that following such reference there is a keyword such as *input* that applies only to that subport. If more than one keyword is needed, they are enclosed in braces, e.g., *{input report}*. Thus we see that all three subports are to come from input, and the **m** subport is to be reported.

The need to make direct subport connections also arises in defining classes that have subports. For example, the **mixerMP** class might be (partially) implemented using the concise syntax as:

```
declare enthalpy  e1, e2, e3;
declare sum s;
declare balance   hb,  wb;
link AirEnt1      .airEnt1,
                  (.TDb) e1.TDb
                  (.w){e1.w, wb.q1}
                  (.h){e1.h, hb.q1}
                  (.m){s.a, hb.m1, wb.m1};
link AirEnt2      .airEnt2,
                  (.TDb) e2.TDb
                  (.w){e2.w, wb.q2}
                  (.h){e2.h, hb.q2}
                  (.m){s.b, hb.m2, wb.m2};
```

[18] This is somewhat like plugging a 2-wire appliance cord into a 3-wire wall outlet.

```
link AirLvg       .airLvg,
                  (.TDb) e3.TDb
                  (.w){e3.w, wb.q}
                  (.h){e3.h, hb.q}
                  (.m){s.c, hb.m, wb.m};
```

Here we see that each subport of the three *macro port*s is linked to the appropriate ports of the constituent enthalpy and balance objects. The normal syntax could also be used here, but this would require four times as many statements.[19]

3.6 Internal SPARK Names for Variables (Full Names of Links or Ports)

In our early examples the name of a problem variable was synonymous with the user-defined name assigned in a *link* or *input* statement. For example, in:

```
declare room r;
link Ta r.Ta;
```

Ta is the *link* name and it obviously represents the variable placed at the **Ta** *port* of the **r** object, probably a room air temperature. However, due to the hierarchical nature of SPARK programming, there are places where *internal* names used by SPARK might not be quite so obvious. This matter can be important when you are reading certain SPARK files, such as the ***.eqs*** file for complex problems, and when using the *probe* keyword (See Section 3.7, page 49).

To understand SPARK naming conventions you must understand that at solution time the solver works entirely at the equation level. This means that when SPARK parses a problem file, all macro objects and macro links must be expanded into atomic objects and links. When this happens, *link* names in higher level objects are propagated downward, as might be expected, overriding names that may have been assigned in the class definition of lower level object. For example, suppose that the **room** class used in the above *link* statement is (partially) defined as:

```
declare  cond      flow; /*  Air mass flow "conductor"    */
declare  cond      walls;/*  Walls conductance            */
declare  cond      floor;/*  Floor to air conductor       */
declare  diff      net;  /*  Diff between Q in and out    */
declare  propcont  pc;   /*  Proportional controller      */
link     Tair .Ta, flow.T2, walls.T1, floor.T1, pc.signal [deg_C];
```

From this we can see that the problem level *link* named **Ta** is known as **Tair** inside the room class, and is connected to the **Ta** *port* of that class, and to ports of various names of the constituent classes of room. By the noted propagation rule, all of these lower level names are overridden by the problem level name **Ta**.

As a result of this downward propagation of *link* names, all problem level variables are readily identifiable when reported, for example, in the *.eqs* file.

However, often there are links in lower level objects that do not appear at the problem level. This occurs whenever a macro class developer elects not to connect an internal *link* to a *port*, or if the user of the class elects not to connect some unessential *port* (i.e., one with the NOERR keyword. See Section 4.9, page 74). As an example, the **mixer** class in the *HVAC Toolkit* class library is defined as:

[19] The **mixerMP** class is one of the many classes in the HVAC Tool Kit implemented in the macro port form.

```
port m      "Combined flow rate, e.g., total mass flow" ;
port q      "Combined transported quantity, e.g., enthalpy" ;
port m1     "First inlet flow rate" ;
port q1     "First inlet transported quantity" ;
port m2     "Second inlet flow rate" ;
port q2     "Second inlet transported quantity" ;
declare     safprod     sp1, sp2, sp;
declare     sum   s;
link        .m, sp.a ;
link        .q, sp.b ;
link   c    sp.c, s.c ;
link        .m1, sp1.a ;
link        .q1, sp1.b ;
link   a    sp1.c, s.a ;
link        .m2, sp2.a ;
link        .q2, sp2.b ;
link   b    sp2.c, s.b ;
```

Note that the links named **a**, **b**, and **c** are not connected to ports. Consequently, they cannot be accessed from higher level objects, and therefore cannot be problem level variables.[20] Nonetheless, these links represent variables whose values must be calculated by the SPARK solver at run time, and they will be assigned names by the SPARK parser. Under normal circumstances, you would not need to know these names; after all, they are merely intermediate variables needed to solve the mixing equations. However, if your problem does not solve properly you may have need to look in the *.eqs* file (Section 2.2.3, page 8), in which case you may want to know the names SPARK assigns to such links. Also, if you need to use the *probe* keyword, you will need to know how to refer to lower level links and ports (See Section 3.7, page 49).

Link names that do not resolve to problem-level links are generated by concatenation of object, link, and port names beginning at the highest level at which the *link* appears and going down to the *port* of an atomic class. The special prefix symbols single quote (`'`), tilde (~), and dot (.) are used in the concatenation in order to ensure unambiguous names. As an example, if we declare a **room** in a problem file as:

```
declare room r;
```

and the room declares a **mixer**:

```
declare mixer mix1;
```

then the **c** *link* in the **mixer** would be referred to as:

```
r'mix1~c
```

This might be read "the **c** *link* in the **mix1** object in the **r** object." The single quote (`'`) prefixes an object in a hierarchy of objects, while the tilde (~) prefixes links. In a more complex situation, objects may be nested deeper, for example,

```
obj1'obj2'obj3~linkname
```

Also, as mentioned in Section 2.5.2 (See page 19), links within a macro class are often unnamed. In this case, SPARK will use a generated string of the form "NONAME*n*" where *n* is an integer. Thus you might see:

```
obj1'obj2'obj3~NONAME7
```

in SPARK *.eqs* files.

An additional complication is introduced when macro links are used (Section 3.5, page 44). Since macro links may have several subports, the linkname must be qualified with the name of the particular *port* of interest. For example,

---

[20] Unless the *probe* statement is used (Section 3.7, page 49).

```
obj1'obj2'obj3~linkname.p1
```

refers to the **p1** *port* of *link* **linkname** in **obj3** that is part of **obj2** that is part of **obj1**. And if the **p1** *port* itself was in fact a *macro port*, we could go on with:

```
obj1'obj2'obj3~linkname.p1.a
```

to refer to the **a** subport of the **p1** *port* of the *link* **linkname** in **obj3** which is part of **obj2** which is part of **obj1**. Fortunately, since you are primarily concerned with higher level problem variables, you don't often have to cope with this complexity.

3.7 Using the Probe Statement

As noted in the preceding section, there are often SPARK links that are not visible at the next higher level due to not having been elevated to a *port* of the class in which they are defined. Yet, sometimes it is convenient or necessary to be able to gain access to such links from higher levels. For example, you may want to report the **c** *link* internal to the **mixer** object in Section 3.6 (See page 47). While you could solve this problem by editing the **mixer** class, i.e., adding a new *port* for **c**, this is not a good solution. First, making changes to widely used classes is hazardous; errors might be introduced, or you might cause unwanted behavior in other applications that use it. Another reason to avoid this approach is that if the needed access is several levels up in a hierarchy, you will have to edit every class in the hierarchy to elevate the needed link to where it is needed. The *probe* statement is provided to give an easier and better solution to such problems. It allows you to reach down into lower level objects, either to report values or set *default*, *init*, *min*, or *max* values. You can also set *match_level* and *break_level* for the *link*.

The *probe* statement has the same general format as the *link* statement. However, you must use the full, SPARK generated, name for the low level *link*, as explained in Section 3.6 (See page 47). As an example, we will use *probe* to set the *init* value and request reporting for the **c** *port* of the **mixer** in the **room** mentioned in Section 3.6 (See page 47):

```
probe mixer_c  r'mix1~c init=0.5 report;
```

This statement would be put in the problem file in which the room **r** is declared. Here **mixer_c** is a user-defined name for the *probe*. The expanded name of the wanted lower level *link* is **r'mix1~c**. With the *init* keyword we set the initial value, to be used if this *link* was selected as a break variable for iterative solution, to 0.5. Finally, the *report* keyword causes the value of c in the **mixer** to be reported along with other requested report variables during solution. The *probe* name **mixer_c** will be used as the label in the requested reporting.

As an aside, it is interesting to note that the above statement could also be written as:

```
probe mixer_c  r'mix1'sp.c init=0.5 report;
```

or as:

```
probe mixer_c  r'mix1's.c init=0.5 report;
```

In these alternative forms, we set the *probe* to point at the **c** *ports* of either the **sp** or **s** objects to which the **c** *link* is connected. Since the values on the *ports* will be the same as the value on the *link* at run time, the same values will be reported.

3.8 Symbolic Processing

As seen in earlier examples, SPARK atomic classes are constructed from equations. While these classes can be constructed manually, the process can be time consuming and tedious. First, the equation must be

solved for all (or most) of its variables, one at a time. For example, if the equation is the ideal gas relationship $pv = nRT$, we need to do the algebra to get the following formulas:

$$p = nRT / v$$
$$v = nRT / p$$
$$n = pv / RT$$
$$R = pv / nT$$
$$T = pv / nR$$

These are called *inverses* of the original equation. Then, for each inverse we must construct a *C++* function that evaluates the right hand side and returns the resulting value. Finally, all of these functions must be incorporated in a SPARK atomic class representing the ideal gas law, following the syntax shown in the earlier examples (Section 2.4.1, page 14).

Fortunately, these tasks can be automated using *symbolic processing* (also called *computer algebra*) tools. SPARK provides a program called *sparksym* that fills this need. With it you can generate all symbolic inverses of an algebraic equation, generate C++ functions implementing these inverses, or create the complete SPARK atomic class.

Actually, *sparksym* is an interface to third-party symbolic programs. Currently, it can use either Mathomatic, Maple, Mathematica or MACSYMA, as selected by a command line option. A subset of the Mathomatic program is integrated in *sparksym*, so that option is always available.[21] If Maple, Mathematica or MACSYMA are detected on your machine when SPARK is installed, or if you install them later and take steps to link them to SPARK, you can select it as an alternative symbolic engine for *sparksym*. Maple, Mathematica and MACSYMA are more powerful than Mathomatic, allowing more complex equations to be handled.

3.8.1 Simple Symbolic Processing

Command-line usage of *sparksym* is with the command:

```
sparksym –engine -option [name] "equation" [target] [outFile]
```

where:

engine = O (Mathomatic), P (Maple), E (Mathematica), S (MACSYMA)
option = i (single inverse), a (all inverses), f (function), c (class)
name = Name for function of class (used only with option f or c)
equation = An equation of the form <expression>=<expression>(enclose in double quotes if spaces occur)
target = The variable to be solved for (used only for options i and f)
outFile = Optional file for the result

[21] The *sparksym* executable provided with SPARK does not give you the full capability of Mathomatic. You can download the DOS shareware program from http://www.lightlink.com/george2/. Among other features, it is capable of symbolic elimination of variables and equations in sets of equations; sometimes this feature can be used to help develop efficient SPARK classes.

3.8.2 Generating an Inverse

For example, to generate the inverse equation for *T* using the ideal gas law, with output to the screen:

```
sparksym –O -i "p*v = n*R*T"  T <enter>
Inverse:
 T = p*v/n/R
```

Or to create the SPARK idealGasLaw atomic class, with results written to idealGasLaw.cc:

```
sparksym –O -c idealGasLaw "p*v=n*R*T" idealGasLaw.cc <enter>
```

The class generated is directly usable, but perhaps not as complete as you may wish. For example, the ports are all assigned a description which is the same as the port name, units are [-] (i.e., unspecified), and the *init*, *min*, and *max* values are set at 1, -100000, and 100000 respectively. You can edit the output file to give more appropriate values for these items if you wish.

3.8.3 Caveats

You are advised to carefully check all symbolic results, since computer algebra software often gives unexpected results, sometimes simply wrong. *Sparksym* using the Mathomatic option is not as robust as a full-featured symbolic package, although it may meet many of your needs. With it, you are limited to expressions using the operators +, -, *, /, and ^ (exponentiation). It will fail quickly if it cannot easily invert the equation for the desired variable. Note that the atomic class generated with the -c option will have functions for each variable in the equation, whether or not an explicit inverse was found for it. Variables for which it could not find an explicit inverse use an implicit inverse as in Section 3.11.2 (See page 60). You may wish to edit the implicit functions, as discussed in the same Section, to improve numerical stability. With the Maple option, practically any equation can be handled, including various mathematical functions. Additionally, it will sometimes find multiple inverses. In this case all inverses are written in the generated functions, with all but one commented out. Therefore it is a good idea to examine the generated class to see that the wanted inverse is being used.

All of the above functionality is also available in the *WinSPARK* and *VisualSPARK* interfaces. See the appropriate Installation and Usage Guide for particulars.

3.9 Previous Value Variables, or Updating Variables from Links

As discussed in Section 3.3.4 (See page 41), most SPARK variables are determined by solution of the problem equations at the current simulation time. This means that each variable gets assigned a value that is calculated from an inverse of one of the problem equations. There are situations, however, when a variable in a simulation must represent the *previous value* of some other variable. Such a variable needs no equation since its value is determined merely by assignment of the value of some variable at the previous point in time. A variable of this nature can be called a *previous value variable*.

Since SPARK variables are carried on *links*, previous value variables are viewed as receiving values by updating from specified links. Consequently, SPARK provides *update_from_link* as an optional keyword in a *link* statement, taking the form:

```
link linkName <connections> update_from_link = FromLinkName;
```

At the beginning of the time step, before solving the problem equations, the saved previous value of **FromLinkName** is assigned to **linkName.** As discussed in Section 3.3.1 (See page 39) initializing a

previous value variable must come from the *init=* keyword in the **FromLinkName**, not in the previous value variable link itself. Indeed, it is an error to place the *init* keyword in a link statement that contains the *update_from_link* keyword. Alternatively, the initial value can come from an *.inp* files as discussed in Section 3.3.1.

As an example we shall revisit the Euler integration formula discussed in Section 2.6 (See page 22). For simplicity there we implemented the *Euler* integration formula as a SPARK atomic class with a single *port* representing the variable of integration, and the name of this *port* was used both as the returned result and in the argument list, i.e.,

```
x = euler(x, xdot, dt);
```

However, this results in unnecessary iteration since the SPARK parser will not know that, internal to the function, only the past value of x is used. We can use the *update_from_link* keyword to correct this deficiency as follows. First, we modify the atomic class to have both current and previous *x* as ports, and properly designate $\dot{x}$ as referring to the previous time value:

```
/*          euler_formula.cc              */
#ifdef      spark_parser
port  x;
port  x_p;
port  xdot_p;
port  dt;
functions {
      x = euler_formula(x_p, xdot_p, dt);
      }
equations {
      x = x_p + dt*xdot_p;
      }
#endif      /*spark_parser*/
#include "spark.h"
double euler_formula(ArgList args) {
      const double& x_p = args[0][1];    // previous x
      const double& xdot_p = args[1][1]; // previous xdot
      const double& dt = args[2];        // time step
      if(::IsInitialTime())
            return args[0].GetInit();
      else
            return x_p + dt*xdot_p;
}
```

Note that we have named this atomic class **euler_formula**. This allows us to define a macro class called **euler** which conceals the complexity of the *update_from_link* considerations and preserves the convenient interface used in the Section 2.6 example. Here is the **euler** macro class:

```
/*          euler.cm             */
port  x;
port  xdot;
port  dt;

declare euler_formula e
link  .dt     e.dt;
link  .x      e.x;
link  X       .x
link  XDOT    .xdot
link  x_p     e.x_p     update_from_link = X;
link  xdot_p  e.xdot_p  update_from_link = XDOT;
```

With this implementation, the ports refer only to current time values of x and $\dot{x}$. Internal to the macro class we create links for both current and previous values of x and $\dot{x}$. The previous value variables, however,

are specified to have their values updated from the corresponding current time values. Note that a *link* name, not a *port* name, must follow the *update_from_link* keyword. Due to this requirement we define two *links* X and XDOT, connect them the *ports* **x** and **xdot**, and use them as arguments to the *update_from_link* keywords. Finally, note that it is not necessary to initialize the previous value variables in this example because, as a consequence of the if-statement in the function definition, they are not used at *InitialTime*.

There are uses for previous value variables other than in integrators for solution of differential equations. For example, simulation of discrete time controllers requires past values, both to calculate controller "integral action" and to determine when to update the controller output. An additional usage is for introduction of an artificial time delay in a troublesome iterative loop. By simply making some variable in the loop a previous value variable the need for iterative solution is removed. If the time step is short, the error introduced may be acceptable.

3.10 Solution Method Control

While the fundamental, graph-theoretic methodology in SPARK is always the same, there are some options you can set to control the actual numerical methods employed. The graphical user interfaces (Windows 95/98/NT or UNIX) provide menus for setting these options. If you are working at the command line, you can set these options by editing the *probName.prf* file. However, to explain these options we must first review the fundamental SPARK methodology.

3.10.1 SPARK Problem Components

As noted previously, SPARK generates a *C++* program to solve the problem expressed in your *probName.pr* file. To generate this program, graph-theoretic methods are used to decompose the problem into a series of smaller problems, called "components," that can be solved independently. A component might be a sequence of atomic-object inverse functions that need to be executed in order; this is the case if no iteration is required in that particular component. On the other hand, iteration may be required, in which case the component, in graph theoretic terms, is a "strongly connected component." While all equations in a strongly connected component are involved in the iterative solution, usually not all variables need be iterates. Therefore SPARK uses graph algorithms to determine a small set of so called "break variables" that break all cycles in the component; these variables constitute a "cut set."

By default, SPARK will attempt to solve each strongly connected component using the Newton-Raphson method, treating the cut set as the vector of independent variables (See Section 3.2, page 38). If your problem solves correctly with the default method, it is probably best not to change it. However, if it fails to solve, it will probably be due to either non-convergence of the Newton-Raphson iteration, or numerical exceptions (i.e., values of problem variables that exceed the capabilities of the computer). In either case, it is usually possible to determine which component is having difficulty by looking at the *probName.log* file or the *run.log* file. You may then want to change the solution method for that component from among the options discussed below.

Solving method options fall into two categories: Component Solving Methods, and Matrix Solving Methods. Component Solving Methods refer either to modifications of the Newton-Raphson method, or a completely different method of finding values for the break variables that satisfy the component equations. Matrix Solving Methods refers to the way in which the next estimates of the break variables are determined from the current values using the Jacobian matrix.

Full explanation of the advanced methods is beyond the scope of this manual. The cited references were consulted in the SPARK implementation.

3.10.2 Default Settings

As noted in Section 2.2.1 (See page 6) every SPARK problem has a *probName.prf*, created by the SPARK setup program. When the problem is executed the solving method settings and associated parameters are taken from this preference file. If you use one of the graphical user interfaces, such as *WinSPARK* or *VisualSPARK*, you can use provided menus for setting the solving methods and parameters, and the settings you specify will be transferred to the problem preference file. If for any reason the preference file does not define a particular method or parameter, default settings built into the source code are used. These default settings are given in the tables below. These are "safe" but not necessarily recommended settings, so you should normally provide appropriate settings for your problem.

3.10.3 Component Solving Methods

The available methods for solving the component are listed in Table 3.1. The code numbers are needed only if you want to set the option by editing the *probName.prf* file. To set the component solving method in the preference file, the *ComponentSolvingMethod* key must be set to the desired code number under the *ComponentSettings* key for the component in question. When using a graphical user interface the available choices are on a selection menu. Note that the solving method chosen will depend on the component. For example, non-iterative components do not need any solution method. Also, when there is iteration, not all methods are applicable for components with more than one break variable. For example, Brent's method applies only when there is a single break variable. You can examine the *probName.eqs* file to see how many break variables there are for each component.

Table 3.1 Component Solving Methods (Not all implemented in initial release)

Method	Code	Notes	Reference
Newton-Raphson	0	With or without relaxation (default).	(Conte and de Boor 1985)
Multi-start ABS	1		(Sen 1994)
Fixed point iteration	2	Successive substitution	
Steffensen acceleration	3		(Press, Flannery et al. 1988)
Secant	4	Multidimensional secant (using Broyden's update formula).	(Press, Flannery et al. 1988) (Dennis and Schnabel 1996)
Homotopy	5	First degree only.	
Brent	6	Valid only for single break variable components.	(Press, Flannery et al. 1988)
All in Turn	7	Try each of the above methods in listed order.	

In addition to the basic solution method for a component, there may be parameters that control how the method behaves. Available control parameters as shown in Table 3.2. For example, with Newton-Raphson method you may want to use "Relaxation," whereby the calculated corrections to the break variables are only partially applied. This is achieved by using a fractional relaxation coefficient. Additionally, in some cases it may be beneficial to "scale" the Jacobian matrix. SPARK allows four different scaling methods.

The default values in the table are used only if the parameter in question is not defined in the *probName.prf* file. However, since a preference file is created automatically for your problem, these defaults are seldom used.

Table 3.2 Component Solution Parameters

Parameter **[*key* in preference file]**	**Allowed values**	**Notes**
Maximum Iterations [*MaxIterations*]	An integer >0	Maximum allowed iterations when iterative solution is used. Default = 50
Tolerance [*Tolerance*]	A floating point number>0.0	Solution relative tolerance. In iterative solution, iteration will continue until no break variable y changes by more than Tolerance*|y| between two successive iterations. Default = 1.E-6
Maximum Tolerance [*MaxTolerance*]	A floating point number > Tolerance	Maximum Tolerance used for a "relaxed" tolerance check instead of Tolerance in case of no convergence after maximum iterations (see Tolerance definition above). Default = 1.E-3
Absolute Tolerance [*AbsTolerance*]	A floating point number > 0.0	Value at which the variable with the smallest order of magnitude is essentially insignificant. Default = 1.E-6
Jacobian Evaluation Step [*TrueJacobianEvalStep*]	Integer >= 1	The Jacobian will be re-evaluated only after this number of iterations. Default = 1
Epsilon [*Epsilon*]	A floating point number >= 0.0	Change in independent variable used in evaluation of partial derivatives for Jacobian calculation. Default = 0 (see Section 3.10.6).

Step Control Method [*StepControlMethod*]	Integer >= 0	Controls the length of the step computed by the component solving method to achieve "global" convergence. 0 = Fixed relaxation (see Relaxation Coefficient); 1 = Basic iterative backtracking attempting to decrease Euclidean norm of residuals; 2 = Backtracking with line search.[22] Default = 0.
Relaxation Coefficient [*RelaxationCoefficient*]	0 < Floating point number <= 1.0	This is a multiplier applied to the Jacobian calculated change to get the actual change during Newton-Raphson iteration. • Fixed relaxation coefficient used with the step control method 0. • With the other step control strategies, this is the relaxation coefficient used to recover when the backtracking method fails to decrease the cost function. Default = 1.0
Scaling Method [*ScalingMethod*]	Integer >= 0	Scales the Jacobian before using. 0 = No scaling; 1 = Curtis-Reid optimum scaling of Jacobian; 2 = Scaling of Jacobian based on right-hand side residual vector; 3 = Scaling of Jacobian based on columns. Default = 0.

3.10.4 Matrix Solving Methods

In Newton-Raphson and related component solving methods a linear set of equations must be solved at each iteration, yielding a correction to the current estimate of the cut set variables. By default, SPARK will use Gaussian elimination to effect this solution. However, other options are available as shown in Table 3.3. The code numbers are needed only if you want to set the option by editing the *probName.prf* file. To set the matrix solving method in the preference file, the *MatrixSolvingMethod* key must be set to the desired code number under the *ComponentSettings* key for the component in question.

[22] (Dennis and Schnabel 1996) should be consulted for more details on the backtracking with line search step control algorithm.

Table 3.3 Matrix Solving Methods

Method	Code	Notes	Reference
Gaussian Elimination	0	Default	(Conte and de Boor 1985)
Singular Value Decomposition (SVD)	1	Poorly conditioned matrix.	(Press, Flannery et al. 1988)
Lower-Upper Factorization (LU)	2		(Conte and de Boor 1985)

In addition to the matrix solving methods shown in Table 3.3, there are also parameters that control their behavior. These are shown in Table 3.4. Note that not all parameters apply to every method.

Table 3.4 Matrix Solving Method Parameters

Parameter **[*key* in preference file]**	**Values**	**Notes**
Pivoting Method [*PivotingMethod*]	0, 1, 2	Only used with the Gaussian Elimination matrix solving method. 0 = No pivoting; 1 = Partial pivoting, row pivots; 2 = Total pivoting, rows and columns.[23] Default=1
Refinement Method [*RefinementMethod*]	0 < Integer < 5	Only used with the LU solving matrix solving method.

3.10.5 Stopping Criterion for Iterative Solution

SPARK employs a mixed absolute/relative tolerance as the stopping criterion used to decide when to terminate the iterative solution in a component. That is, for a break variable *y*, the convergence criterion is that the iteration error satisfies:

$$Error(y) \le \max\left(AbsTolerance, Tolerance \cdot |y| \right) \tag{3.6}$$

The value of *Tolerance* is specified with the key "Tolerance" in the problem preference file on a per-component basis. The value of *AbsTolerance* is specified with the key "AbsTolerance" for the component in question. By default, it should be set to *Tolerance* unless the variables have very different orders of magnitude.

[23] The Gaussian elimination solving method with full pivoting is also referred to as the Gauss-Jordan elimination solving method in (Press, Flannery et al. 1988).

Such a scaled tolerance requirement is necessary to achieve convergence with a consistent number of significant digits, *p*, for variables with different orders of magnitude. The relationship between the tolerance and the number of significant digits in the solution is:

$$Tolerance = 10^{-(p+1)} \tag{3.7}$$

Clearly, it is important to carefully select the error tolerance setting for each component so as to accurately reflect the scale of the problem. For components whose break variables are scaled very differently from each other, the *AbsTolerance* value should be set to the value at which the break variable with the smallest order of magnitude is essentially insignificant. This should ensure that the variable with the smallest scale does not limit the accuracy with which the other variables are computed. When all the break variables are of comparable order of magnitude and their values are not near the *AbsTolerance* value, then the *Tolerance* value gives an indication on the number of significant digits in the solution, using Equation (3.7). In this case, Equation (3.6) tends to enforce a pure relative tolerance requirement. However, if the values of the break variables are near the *AbsTolerance* value, then you should not expect the relation in Equation (3.7) to hold precisely. In this case, Equation (3.6) tends to enforce a pure absolute tolerance requirement.

3.10.6 Scaled Perturbation for Partial Derivatives

In SPARK, Newton based iterative solution methods (i.e., Newton-Raphson and Homotopy) require the Jacobian matrix to be computed. This matrix consists of the partial derivatives of the iterated system of equations with respect to the break variables. These partial derivatives are approximated by finite differences. For example, the partial derivative of the equation $f(t, x, y)$ with respect to the break variable y is approximated using the following formula:

$$\frac{\partial f(t,x,y)}{\partial y} \approx \frac{f(t,x,y+\Delta y) - f(t,x,y)}{\Delta y} \tag{3.8}$$

Here Δy is called the perturbation value, or increment, of the variable *y*. You can specify the value of the perturbation value for each component using the keyword *Epsilon* in the problem preference file (See Section 3.10.3, page 54).

The differencing procedure in digital computation is sensitive to roundoff error. The main source of difficulty in computing the Jacobian matrix by finite differencing is the choice of the perturbation Δy. Consequently, SPARK provides the option to use a *scaled* perturbation value to compute the partial derivatives. This is done by specifying a zero value for the *Epsilon* component setting in the preference file for the component in question. For example, if you wish to use scaled perturbation in Component 0, the preference file should include:

```
ComponentSettings (
      0     (
            RelaxationCoefficient ( 1.0 ())
            Epsilon ( 0 ())
            Tolerance ( 1e-6 ())

            …
            )
)
```

When *Epsilon* is specified as zero, SPARK computes the perturbation value for the variable *y* as:

$$\Delta y = sign(\dot{y}) \cdot \max\left(|y|, |y + h \cdot \dot{y}|, Tolerance \right) \cdot \sqrt{URound} \tag{3.9}$$

Here, *URound* is the machine unit round-off error. The derivative, $\dot{y}$, with respect to the independent variable (usually time) is approximated using the explicit Euler scheme. The term $|y + h \cdot \dot{y}|$ is included to represent the predicted value for *y* at the next step. This is because even if $|y|$ happens to be near zero, it is quite possible that a nearby value of *y* is not so small, and selecting $|y + h \cdot \dot{y}|$ will prevent a near zero perturbation from being used. In the event that $|y|$ and $|y + h \cdot \dot{y}|$ are both near zero, the error tolerance *Tolerance* is used as a lower bound in the formula to prevent using too small a perturbation. Indeed, by setting the error tolerance, you tell the SPARK solver that it is the smallest number which is relevant with respect to the break variables *y* in this component.

The formula in Equation (3.9) perturbs about half of the digits of the variable *y* when *y* is significantly larger than *Tolerance*. Finally, note that the sign of the perturbation Δy computed with Equation (3.9) will be negative if the solution is decreasing. Unfortunately, this choice is a potentially source of difficulty for problems where some functions are undefined for $y < 0$ or not differentiable at $y = 0$.

3.10.7 Update Component Settings at Run Time

In some situations you may need to change some parameters of the component settings at run-time. To support this need, SPARK optionally checks the time stamp of the problem preference file while executing, and when it changes the file is read again, loading the new component settings. These settings become effective for the next time step. However, in SPARK 1.0, only the following parameters can be updated at run-time using this mechanism:

- Relaxation coefficient
- Tolerance
- Epsilon

To allow updating component settings at run time, the following entry should appear in the run control file:

```
UpdateComponentSettingsAtRunTime ( 1 ( ) )
```

If this is not specified, the parameters of the component settings will not be updated at run-time.

These controls give you control over the convergence process, which may be important for large nonlinear problems requiring long run times. In order to determine if and when you need to change the settings, you should set the diagnostic level 1 or 3 (see Section 3.11.5, page 63) to be able to follow interactively the convergence process.

3.11 Debugging SPARK Programs

Often SPARK will find calculation sequences leading to successful problem solution without intervention. However, solution of nonlinear differential and algebraic equations is not easy, even for SPARK, and in some cases you may get error messages. These may be during the initial processing where your input is being parsed, while executing the setup program that converts it to a solver program, or during execution of the solver program, i.e., at run time.

3.11.1 Parsing Errors

Parsing errors are usually syntax errors, as in any programming language. These errors are reported in the *parser.log* file, normally placed in your project directory. They should be easy to interpret, but if not the command reference in Section 4 (See Page 71) may be helpful.

3.11.2 Setup Errors

During the setup phase SPARK may have other difficulties due to input errors. For example, you may have specified a problem for which no matching can be found between equations and variables. This can happen even if you have an equal number of equations and free variables (i.e., *links*). As an example of this, consider the *4sum* problem when *x1, x5, x6,* and *x7* are specified as *input*s. This is not well posed because it over-determines the equation for *s3* while under-determining *s2*. SPARK will report such errors as "unable to find a matching." Subtle errors of this nature can occur in development of complex models. Setup errors are reported in either *setup.log* or *probName.log*, depending upon your platform.

Unfortunately, lack of matching can also arise for well posed problems if you have not provided enough inverses for your atomic objects. Complex models involve equations that maybe difficult to invert, even with symbolic algebra tools. Consequently, it is common for SPARK users to omit the difficult inverses for some equations, providing only those easily come by. Usually, this is acceptable practice since SPARK explores many paths to a get a solution sequence and usually finds one. However, if you are experiencing matching problems and have omitted some inverses you may want to consider using *implicit* inverses. For example, if you cannot solve $g(x,y,z) = 0$ for x, simply write for the inverse

$$x = f(x, y, z)$$

where $f(x,y,z)$ is an algebraic rearrangement of $g(x,y,z)$ that is as far as you can go in isolating x. Best numerical performance will be obtained if $f(x,y,z)$ is only weakly dependent upon x. However, if all else fails, simply write:

$$x = x + g(x, y, z)$$

SPARK will discover that x is on both sides of this "inverse" and place it in the cut set, in effect inverting the troublesome equation numerically.

3.11.3 Solution Difficulties

Even after SPARK has successfully created a solver program there can be difficulties in finding a solution. This is because of the nature of nonlinear systems of equations, with which numerical analysts have been struggling for many years. Here we are referring to convergence difficulties; the solver iterates the maximum allowed number of times (set by default to 50) without bringing the solution into the error tolerance (default 1.e-6). If you work with complex systems, resolving these difficulties is the greatest challenge you will face. Run-time errors are reported to *run.log* or *probName.log*, depending upon your platform. More detailed error messages and diagnostic can be found in *error.log*.

With SPARK, you attack convergence problems in two basic ways: estimating better values to start the iteration, and by trying to alter the solution sequence. The importance of good iteration initial values is well known; in this regard, the only difference between SPARK and other simulation tools is with SPARK, due to reduction in the number of iteration variables, you do not have to specify as many guess values. We discuss how to set initial iteration values in Section 3.3.2 (See Page 40).

The second strategy, controlling the solution sequence, is based on the observation that iteration can usually be done many different ways, often differing in the direction in which calculations flow around cycles in the problem graph. Sometimes convergence can be achieved by calculating in the opposite direction. Consequently, SPARK provides syntax in the definition of problems and classes in order to control, indirectly, the calculation direction. You can always see the solution sequence chosen by SPARK in the *.eqs* file produced by the setup program. Open this file with a suitable viewer or editor and use it as guide in understanding and improving your problem solution sequence.

Match_level is very effective in reversing the direction of calculations in SPARK. By default, matchings are found based only on order of objects and links found in the problem specification file. By forcing or encouraging a different matching you can often improve numerical performance, and perhaps achieve convergence.

The relevant keywords are *match_level* and *break_level*. Each can be set to a value between 0 and 10. When left unspecified, these levels default to 5. The *match_level* keyword is placed in a *link* or port statement, and specifies the relative desirability of matching that link variable to a particular object in the *link* statement. For example,

```
link  x  a_obj.p1 match_level = 10, b_obj.p3;
```

tells SPARK that you would prefer that object **a_obj** should be matched with the **x** problem variable. You could say somewhat the same thing by the statement

```
link  x  a_obj.p1, b_obj.p3 match_level = 0;
```

which says you would prefer that **x** not be matched with object **b_obj**. Provided that you not simply encourage selection of the matching that would be found by default, the direction of calculations in the problem will be reversed. Currently, the second form is stronger that the first due to the implementation of the matching algorithm used in SPARK.

Break_level parallels the *match_level* idea, but applies to the discovery of a cut set, i.e., selection of variables to break cycles in the problem graph. When there is a cycle, usually many problem variables are encountered as you work your way around the loop. It is easy to see that any of these variables will break the loop. By default, SPARK sets break preference to 5 for all variables, so the break selected is determined solely by order in the problem definition. Yet, there are sometimes arguments for preferring one over another.

A simple example is based on starting value availability. If you have the choice of breaking on enthalpy or temperature, you may prefer the latter simply because you are likely to be able to better estimate iteration starting values for temperature. Some analysts also feel that different break variables lead to better convergence. However, the "gain" around the loop is going to be the same regardless, so this may not be a strong argument. Nonetheless, if you have any reason or hunch that a particular variable would be a better break, give it a high *break_level*. To do so, include it in the *link* statement:

```
link  x a_obj.p1 break_level = 7, b_obj.p3 match_level = 10;
```

In the current implementation, matching and break levels only encourage SPARK to match or break the way you wish. This is because we wanted to give SPARK maximum opportunity to find solution sequences, and denying certain matchings and breaks may prevent any solution at all. In later versions we may also provide forced matchings and breaks.

Finally, it should be noted that these are only indirect tools, sometimes having little or no effect on the solution sequence. For example, setting *break_level* on a link that does not happen to be in a cycle will have no effect, and as already noted setting a *match_level* to force a match that is selected by default is also ineffective.

3.11.4 Trace File Mechanism

Sometimes it may be helpful to see intermediate results of the iterative solution process. This is especially important when your problem is experiencing convergence difficulties. You can get such output by using the *TraceFiles* segment under the key *ComponentSettings* for the component in question in the *probName.prf* file. This is done for individual components (See Section 3.10.1, page 53). As with solution control parameters (See Section 3.10, page 53), setting this flag is done most conveniently with the aid of a SPARK graphical user interface. Otherwise, you can edit the *probName.prf* file directly with any text editor.

The *TraceFiles* segment has five allowed values as shown in Table 3.5.

Table 3.5 Keys and Values for TraceFiles Segment

TraceFiles Key and Value	Meaning
( )	No trace output.
Jacobian (fileName ())	Jacobian of residual functions printed whenever it is recomputed.
Increments (fileName ())	Increments of all variables printed at every iteration.
Residuals (fileName ())	Break residuals printed at every iteration.
Variables (fileName ())	All problem variables printed at every iteration.

Within each component, you can specify up to four trace files entries with the name of each file preceded by one of the keys listed in Table 3.5. Each key specifies the type of the trace file that will be written to the file following the type key. For example, the following segment could be inserted in ComponentSettings 0 of a problem preference file:

```
ComponentSettings (
      0     (
                  TraceFiles (
                        Jacobian   ( spring_jac.trc ())
                        Increments ( spring_inc.trc ())
                        Residuals  ( spring_res.trc ())
                        Variables  ( spring_var.trc ())
                  )
                  …
      )
)
```

Any file name with the extension *.trc* can be used, except it cannot be repeated. That is, you cannot use the same file name for tracing in the same component, or in a different component.

If no trace files are wanted, the *TraceFiles* segment for the component should be:

```
      TraceFiles ( )
```

3.11.5 Problem-level Diagnostic Reports

In addition to the Trace facility (See Section 3.11.4, page 62) SPARK has a problem-level diagnostic facility. To use this feature, the *DiagnosticLevel* keyword must be set to something higher than 0 in the problem run control file (See Section 2.2.1, page 6). Three different modes trigger increasing level of diagnostic to the *cout* stream. When running the *runspark* command, the output goes to the *run.log* file. The default mode is the silent mode.

Table 3.6 Problem-level Diagnostic Flag Values

Mode	Entry in run file	Description
Silent (default mode if no DiagnosticLevel is specified)	DiagnosticLevel (0 ())	Outputs run control parameters, input data, output and snapshot files if any specified
Report convergence	DiagnosticLevel (1 ())	At each iteration, the convergence progress is reported for each component. Includes scaled residuals' norm, convergence error, requested tolerance, name and value of the worst-offender variable.
Report results	DiagnosticLevel (2 ())	All variables are reported with their names and values at each step.
Report convergence + Results	DiagnosticLevel (3 ())	Combines level 1 and 2.

3.12 Output and Post Processing

When SPARK runs there is output to the screen and to an output file with extension *.out*. The screen output is primarily for visual feedback, letting you know where SPARK is in processing your problem. The output file contains results of the numerical solution process at each time step. The format of the output file is exactly like that of input files, i.e.,

```
n       label   label   label
t0      value   value   value
t1      value   value   value
etc.
```

where *n* is the number of reported variables, each *label* is a problem variable with the *report* keyword expressed in the problem file, and each *value* is the value for the corresponding variable at time t_i.

The output of SPARK can be read by conventional spreadsheet and plotting programs. If you use Microsoft Excel or a similar program, simply open the SPARK output file into a worksheet and use tabs as the delimiting character between fields. This will place your output neatly into rows and columns, from which

you can construct plots (charts) in the usual Excel manner. If you use *gnuplot*, a program called *makegnu* is provided with *WinSPARK* that will generate an input file for that program.[24] To use *makegnu*, type:

```
makegnu room_fc.out room_fc.plt <enter>
```

The output file, *room_fc.gnu*, will contain the *gnuplot* commands, e.g.:

```
set data style lines
set xlabel "time"
set ylabel "mcp"
plot "room_fc.out"  using 1:2 notitle
pause -1 "Press <enter>"
set ylabel "Q_flow"
plot "room_fc.out"  using 1:3 notitle
pause -1 "Press <enter>"
set ylabel "Ta"
plot "room_fc.out"  using 1:4 notitle
pause -1 "Press <enter>"
set ylabel "T_floor"
plot "room_fc.out"  using 1:5 notitle
pause -1 "Press <enter>"
```

Then to plot with *gnuplot*, type

```
gnuplot room_fc.plt <enter>
```

This assumes you have *gnuplot* in your command path.

More elaborate plots, combining several results on the same plot, for example, can be done by editing the *gnuplot input* file, or by running *gnuplot* interactively. The *gnuplot* documentation should be consulted for more information.

3.13 Snapshot Files and Restarting Solutions

There are occasions on which you may want to stop a simulation, then restart it from the same point at a later time. This need can arise when the problem experiences a long run time, or a difficult solution. Or, you may want to repeat a simulation using precisely the same initializations of dynamic and break variables. These techniques are supported in SPARK with the notion of snapshot files. You can request that snapshot files be generated at *InitialTime* and/or *FinalTime* as discussed below.

A snapshot file contains the values of all problem variables in a format identical to that of a normal output report. And, because SPARK input files and output files have the same format, you can specify a snapshot file as an input file in a subsequent run of the same problem.

You request generation of snapshot files by specifying corresponding keys in the run control file (See Section 3.14, page 65), along with the desired name for the snapshot file. Two keys are available, *InitialSnapshotFile* and *FinalSnapshotFile*. The values of these keys should be paths to the files where you want the results saved. For example, if you want both initial and final snapshot files, your run control file *probName.run* must contain the following two clauses:

```
InitialSnapshotFile ( probName.init ())
FinalSnapshotFile   ( probName.snap () )
```

[24] Although not provided in the *VisualSPARK* release, *makegnu* is available free from Ayres Sowell Associates, Inc. and will run on UNIX as well as Windows platforms.

InitialSnapshotFile generates a snapshot file with the initial time solution in *probName.init*, whereas *FinalSnapshotFile* generates a snapshot file with the solution at the final time in *probName.snap*. Note that the file names, including the extensions, are arbitrary, i.e., you can use whatever extension you wish. Normally, you will want to include the file path to specify where it is to be saved. In the example, it is saved in the current working directory.

To use a snapshot file for initializing a subsequent run you simply specify it in the *InputFiles* clause in the run control file, with the other input files. For example, to restart your problem initialized from the final solution of the previous run, captured in *probName.snap*, in *probName.run* modify the *InputFiles* clause to read:

```
InputFiles (
      probName.snap ()
      probName.inp ()
)
```

Another way to use snapshot file to restart a problem is to first solve a static problem (no integrators) derived from the dynamic problem and with initial conditions for some of the unknowns of the dynamic problem. This is mimicking what we'll do automatically in SPARK 2. The result snapshot file of the solution of the static problem can then be used to start the dynamic problem with the desired initial conditions enforced.

A snapshot file contains the values for all the problem variables, not just those that were tagged with the *report* keyword in the problem definition file. This means that a snapshot is a very powerful reporting and diagnostic mechanism as well as serving as restart initialization files. Note that if *FinalSnapshotFile* was specified, in the event of a nonconvergence or other solution failure, then the snapshot file will be generated at the time where failure occurred. This provides values of all variables at the point of non convergence, which might be helpful in discovering the reasons for non convergence.

3.14 Run Control File

We introduced the SPARK run control file, *probName.run*, in the Section 2.2.1 examples (See page 6). There, we were concerned with only the basic, required elements of this file needed to run simple problems. In this Section we will examine the run control file further, showing the format as well as all aspects of a SPARK run that can be controlled from it.

The run control information needed for a SPARK problem comprises eleven keys and values as shown in Table 3.7. Items shown in boldface are required.

Table 3.7 Run Controls

Key	Definition	Typical value
InitialTime	The time at which the simulation begins.	0.0
FinalTime	The time at which the simulation ends.	0.0
TimeIncrement	The time between solution points.[25]	1.0
FirstReport	The time at which the first output is desired.	0.0
ReportCycle	The time interval between output reports.	>= TimeIncrement
DiagnosticLevel	Level of diagnostic output desired.	0
InputFiles	List of input file paths.	probName.inp c:\Phoenix\weather.inp
OutputFile	Output file path.	probName.out
UpdateComponentSettingsAtRunTime	Set to 1 to allow updating component settings at run time (only Tolerance, Epsilon and RelaxationCoefficient can be updated)	0 (default)
InitialSnapshotFile	Initial time snapshot file path.	probName.init
FinalSnapshotFile	Final time snapshot file path.	probName.snap

This information is stored in the file *probName.run* using the preference file format, as described in Appendix B (See page 87). A typical run control file is then:

```
(
InitialTime                    ( 0.0 ())
FinalTime                      ( 5.0 ())
TimeIncrement                  ( 0.1 ())
FirstReport                    ( 0.0 ())
ReportCycle                    ( 0.1 ())
DiagnosticLevel                ( 3 ())
InputFiles                     ( frst_ord.inp ()
                                 frst_ord_ic.inp ()
                               )
OutputFile                     ( frst_ord.out ())
InitialSnapshotFile            ( frst_ord_dyn.init ())
FinalSnapshotFile              ( frst_ord_dyn.snap ())
UpdateComponentSettingsAtRunTime ( 1 () )
)
```

[25] In SPARK 1.0, the stepsize or time increment is constant during the course of the simulation. Future versions of SPARK will support variable time-stepping.

3.15 Using SPARK library functions in an atomic class

In Section 3.1 (See page 37), we introduced the boolean function *IsInitialTime()* used in the implementation of the Euler intergator class. The function *IsInitialTime()* is a global function that belongs to the SPARK library. The SPARK library functions that can be called from within the inverse of an atomic class fall into four categories: error handling functions, access functions, predicate functions and math functions. The C++ prototypes for the SPARK library functions can be found in the header files in the directory *vspark\inc*. To be able to use the SPARK library functions in an atomic class, the header file *spark.h* should be included in the file using the C preprocessor *#define* directive.

3.15.1 Error handling functions

These library functions provide support to handle errors from within an atomic class.

The library function *WriteToErrorLog()* lets you write a message to the error log file, called *error.log*. The prototype for this function is:

```
void WriteToErrorLog(
      const char* strFileName,    // name of the atomic class
      const char* strInverseName, // name of the inverse function
      const char* strMsg);        // message to write out
```

For example, in the class *effprl.cc* defined in the HVAC toolkit, when the reference heat exchanger effectiveness *eff* becomes greater than one, we reset the value to a "typical" value and write an error message to the error log to notify the user. This can be done using the following code snippet.

```
double ntuprl ( ArgList args )
{…
    char ErrMsg[100];
    sprintf( ErrMsg, "eff = effp * cRatio = %lf must be <= 1.0\n",
            eff);
    ::WriteToErrorLog( __FILE__[26], "ntuprl()", ErrMsg);
…
}
```

If you wish to stop the execution of the simulation from within an atomic class, you should use the function *ExitFromAtomicClass()* [27] to ensure proper destruction of the objects instantiated by the SPARK solver. The argument list and the usage of this function is similar to the one of the function *WriteToErrorLog()*.

---

[26] __FILE__ is the name of a predefined ANSI C macro that the compiler/preprocessor expands to a C-string (i.e. an object of type *char**) containing the name of the current source file.

[27] Avoid calling the C library function *exit()* directly from an atomic class as this does not allow SPARK to carry out necessary cleanup tasks. The SPARK library function *ExitFromAtomicClass()* calls in turn the C function *exit()*.

```
void ExitFromAtomicClass(
      const char* strFileName,    // name of the atomic class
      const char* strInverseName, // name of the inverse function
      const char* strErrMsg);     // description of error
```

3.15.2 Predicate functions

These library functions return True or False depending on the state of the simulator.

```
bool IsInitialTime();
bool IsFinalTime();
```

3.15.3 Access functions

These library functions provide read-only access to some internal variables of the SPARK solver.

```
unsigned GetStepCount();
double   GetClock();
double   GetStepsize();
```

3.15.4 Math functions

These library functions provide basic mathematical operations that are not part of the C/C++ math library defined in the header file *<math.h>*.[28] They are used in the implementation of the SPARK HVAC toolkit. These functions are self-explanatory.

```
double min(double , double );
double max(double , double );
double sign(double );
```

3.15.5 Access methods for the *TArgument* class

The SPARK inverses are C functions that expect a list of arguments of the type *TArgument*. The *TArgument* type is implemented as a C++ class. The class declaration can be found in the header file *vspark\inc\value.h*. The list of methods that can be used with a *TArgument* object is shown in Table 3.8.

Table 3.8 List of access methods for an object of the TArgument class

Method prototype	**Description**
`operator TArgument::double() const`	Returns the current value of the link as a *double*
`operator TArgument::GetVal() const`	Same as previous method.

[28] In the C++ standard library, the header files assumed from the C standard now have the new prefic *c* instead of the old extension *.h*, i.e., *#include <math.h>* becomes *#include <cmath>*.

`double TArgument::operator[](int idx) const`	Returns the *idx* (*idx>0*) past value of the link as a *double*
`double TArgument::GetInit() const`	Returns the initial value of the link as a *double*
`double TArgument::GetMin() const`	Returns the min value of the link as a *double*
`double TArgument::GetMax() const`	Returns the max value of the link as a *double*
`char* TArgument::GetName() const`	Returns the name of the link as a C-string[29]
`char* TArgument::GetUnit() const`	Returns the unit of the link as a C-string

[29] We call a C-string an object of type *char** .

Section 4 SPARK Language Reference

4.1 Notation Used in this Section

1. Keywords are shown uppercase, although they are case insensitive in the language.
2. ♦ means required syntax.
3. val_or_par means any value, or a parameter name.
4. Items separated by | means choose one of the items (e.g., <x|y|z> means x or y or z)
5. Items inside question marks, e.g., ?connections1?, are defined later in the construct in which they appear.
6. When referring to hierarchy, the problem is called the highest level, while the atomic class is the lowest.

4.2 Special characters

Special characters are those used by the SPARK parser to identify parts of the language. They should not be used in user names.

1. Used in SPARK syntax: " # ( ) , . ; = [ ] ' ` { } ~ /* SPACE TAB NL (newline)
2. Delimiters: SPACE TAB NL. More than one of these characters or combination are ignored.
3. The statement terminator is the semicolon (;).

4.3 Names and Other Strings

4.3.1 Reserved Names

#endif	#ifdef	ABSTRACT

ABSTRACT_END	BAD_INVERSES	BREAK_LEVEL
CONNECT_HINT	DECLARE	DEFAULT
EQUATIONS	FUNCTIONS	GLOBAL_TIME
GLOBAL_TIME_STEP	INIT	PRED_FROM_LINK
INPUT	KEYWORDS	LINK
MATCH_LEVEL	MAX	MIN
NOERR	PARAMETER	PAST_VALUE_ONLY
PORT	PRED	PROBE
REPORT	SIMULT_OUT	UPDATE_FROM_LINK
VAL		

Notes: Reserved names are case insensitive, except for #ifdef and #endif.

4.3.2 Rules for User Specified Names

1. They must not contain any reserved characters.
2. They must not begin with a digit.
3. They are case sensitive.
4. They may not be the same as reserved names.
5. They can be of any length.

4.3.3 Literals

User specified literal strings are enclosed inside double quotes, e.g., "This is a literal". They can contain any character except the double quote (").

4.4 Comments

There are two kinds of comments:

1. /*comment...*/ C-like comment
2. //comment... C++ style comment to end of line

4.5 Compound Statements

A compound statement are delimited by curly braces: { … }. Examples of compound statements are FUNCTIONS and EQUATIONS.

4.6 Atomic Class File

The SPARK atomic class is the smallest modeling element. Atomic classes may be combined in macro classes to form larger modeling elements, or used directly in problem files.

File name convention : *class_name.cc*

Format:

```
/* CLASS  class_name   "description..."
   KEYWORDS=keyword1,...;
   ABSTRACT
*/
#ifdef SPARK_PARSER ♦   if file contains C++ functions
PARAMETER  statements
PORT       statements     ♦
EQUATIONS { equation statements }
FUNCTIONS { function statements }     ♦
#endif /*SPARK_PARSER*/    ♦   if file contains C++ functions
#include "spark.h"    ♦   if file contains C++ functions

   Inverse C++ functions go here.
```

Notes:

1. PARAMETER statements must appear before they are referenced.
2. PORT statements must appear before EQUATIONS and FUNCTIONS statements.
3. While the material in the /*...*/ header is ignored by the parser, it may be used by browsers and/or utility programs.

4.7 Macro Class File

A SPARK macro class connects atomic and other macro classes to form larger modeling elements.

File name convention: *class_name.cm*

Format:

```
/* CLASS_MACRO  class_name   "description..."
KEYWORDS=keyword1,...;
ABSTRACT
*/
PARAMETER  statements
PORT       statements     ♦
PROBE      statements
DECLARE    statements     ♦
LINK       statements     ♦
```

Notes and restrictions:

1. PARAMETER statements must appear before they are referenced.
2. PORT statements must appear before any DECLARE or LINK statements.
3. DECLARE statements must appear before any LINK statements that refer to the objects defined by DECLAREs.

4. While the material in the /*...*/ header is ignored by the parser, it may be used by browsers and/or utility programs.

4.8 Problem File

The SPARK problem file combines macro and/or atomic classes to form the largest modeling element.

File name convention: *problem_name.pr*.

Format:

```
/* PROBLEM  class_name   "description..."
KEYWORDS=keyword1,...;
ABSTRACT
*/
PARAMETER   statements
PROBE       statements
DECLARE     statements      ♦
LINK        statements      ♦
INPUT       statements      ♦ [30]
```

Notes:

1. PARAMETER statements must appear before they are referenced.
2. DECLARE statements must appear before any LINK statements that refer to the objects defined by DECLAREs.
3. While the material in the /*...*/ header is ignored by the parser, it may be used by browsers and/or utility programs.

4.9 PORT Statement

The PORT statement describes an externally visible connection point (interface variable) of a class. When an object is instantiated from a class by a DECLARE statement, the connections can only be made to its ports.

The PORT statement has two forms :

1. Atomic port, which does not have subports.
2. Macro port, which has subports.

An atomic port has the form:

```
PORT port_name    ♦
            [unit]
            "description..."
            CONNECT_HINT="-class1.portx, class2.porty"
            NOERR
            DEFAULT=val_or_par1
            INIT=val_or_par2  MIN=val_or_par3  MAX=val_or_par4
            BREAK_LEVEL=val_or_par5  MATCH_LEVEL=val_or_par6 ;
```

[30] Alternatively, LINK statements with INPUT keyword can be used.

Here:

1. port_name: Name of the port; must not contain any reserved characters.
2. [unit]: Unit of the port. It is used to give a warning if variables with different units are linked.
3. "description...": Short description of the port. This field is used by browsers.
4. CONNECT_HINT: Used by browsers to determine acceptable connections. "-class1.portx, class2.porty" means that connecting this port to portx of any instance of class1 is *not* permitted, but connecting this port to porty of any instance of class2 *is encouraged*. For acceptability, first units, then CONNECT_HINTs are checked.
5. NOERR: Do not give error message if this port is not connected when this class is used (instantiated). Allows ports that can be optionally used.
6. DEFAULT: If this port is not connected, behave as if this value is fixed at val_or_par1.
7. INIT, MIN, MAX: Initial, minimum, and maximum values assigned to variable created by connections to this port. Higher level settings will take precedence.
8. BREAK_LEVEL, MATCH_LEVEL: The default break_level and match_level values for connections to this port.

A macro port has the form:

```
PORT port_name ♦
               [unit1]
               "port description..."
               CONNECT_HINT="-class1.portx,class2.porty"
               NOERR
               .subport_name1    ♦
               [unit2]
               "subport description..."
               DEFAULT=val_or_par1
               INIT=val_or_par2  MIN=val_or_par3  MAX=val_or_par4
               BREAK_LEVEL=val_or_par5  MATCH_LEVEL=val_or_par6
         ,     .subport_name2
         etc.
         , ...
         ;
```

1. Port_name: Name of the port; must not contain any reserved characters.
2. [unit1]: Unit of the port. It is used to give a warning if variables with different units are linked.
3. "description...": Short description of the port. This field is used by browsers.
4. CONNECT_HINT: Used by browsers to determine acceptable connections. "-class1.portx, class2.porty" means that connecting this port to portx of any instance of class1 is *not* permitted, but connecting this port to porty of any instance of class2 *is encouraged*. For acceptability, first units, then CONNECT_HINTs are checked.
5. NOERR: Do NOT give error message if this port is not connected when this class is used (instantiated).
6. .subport_name: Name of the subport. Note the leading dot (.). If subport contains other subports, this is specified as .subport_name.subport_of_subport_ Note that subport_of_subport_name is specified for subport_of_ . For example, if we have port x with subports a, b and subport a has its subports a1,a2 we write:

```
PORT  x  ...etc.
        .a.a1  ...etc.
                , .a.a2  ...etc.
                , .b  ...etc.  ;
```

7. DEFAULT: If this subport is not connected, behave as if this value is fixed at val_or_par.
8. INIT, MIN, MAX: Initial, minimum, and maximum values assigned to variable created by connections to this port. Higher level settings will take precedence.
9. BREAK_LEVEL, MATCH_LEVEL: The default break_level and match_level values for connections to this subport.

4.10 PARAMETER Statement

The PARAMETER statement is used to assign a numeric or symbolic value to a name. When this name is used in any place that can take the parameter name, the value of the parameter is substituted in place of the name. For example the following two statements:

```
PARAMETER abc = 12.3 ;

PORT x  INIT=abc ;
```

have the effect:

```
PORT x  INIT=12.3 ;
```

The parameter statement has the form

PARAMETER name1 = substitution_value1 , name2 = substitution_value2 , ... ;

If a problem and one of its classes have parameters of the same name, the value of the problem's parameter is used. Similarly, if a macro and one of its classes have parameters of the same name, the value of the macro's parameter is used. That is, higher level PARAMETER definitions take precedence.

4.11 PROBE statement

Without PROBE, lower level links (e.g., in a macro object) are not visible at higher levels (e.g., a problem file) unless they are connected through ports. The PROBE statement is provided to allow assigning values to certain keywords for lower level links from a higher level. It can also be used to report such links. See Section 3.7 (page 49) for examples.

The PROBE statement has the form

```
PROBE  name    <?port_resolution? | ?link_resolution?>   ♦
               INIT=val_or_par2  MIN=val_or_par3  MAX=val_or_par4
               BREAK_LEVEL=val_or_par5  MATCH_LEVEL=val_or_par6
               INPUT  REPORT
               PRED_FROM_LINK=<?port_resolution? | ?link_resolution?>
               UPDATE_FROM_LINK=<?port_resolution? |
?link_resolution?>
               VAL=val_or_par;
```

Here:

1. name: Name of probe.
2. ?port_resolution?: Concatenated object name followed by port.subport name that uniquely identifies the connection. It has the form:
   obj1`obj2...port.subport.subport_of_subport...

3. ?link_resolution? : Concatenated object name followed by link name followed by subport of link that uniquely identifies the link. It has the form:
   obj1`obj2...~link.port.subport.subport_of_subport...
   For problem level links this has the form ~link5.subport. subport_of_subport…
4. INIT, MIN, MAX, BREAK_LEVEL, MATCH_LEVEL, INPUT, REPORT, UPDATE_FROM_LINK, PRED_FROM_LINK, VAL: Same as for LINK statement.

4.12 DECLARE statement

The DECLARE statement is used to instantiate a class, creating one or more objects. It has the form

```
DECLARE  class_name   obj_name1, obj_name2, ... ;
```

Here obj_name can be either a valid name or a PARAMETER name that defines a valid name.

4.13 LINK statement

The LINK statement is used to make connections between ports of objects instantiated in this class and/or port(s) of this class. It has the form

```
LINK  name  "link_description"   ?entries1? , ?entries2? , ...
              , (.sublink1...){ ?entries3? , ?entries4? , ... }
              , (.sublinkN...){ ?entriesM? , ... } ;
```

The optional (.sublink1...){ ... } form means that the entries inside{} apply to the .sublink1... component of the macro-link. Here, .sublink... is a valid .portal... name for this link. The ?entriesX? contains items from the following, where at least the ?connection? item must be present:

```
< INPUT
< REPORT
       < GLOBAL_TIME | GLOBAL_TIME_STEP >
< VAL  = val_or_par >
< INIT = val_or_par >
  < MIN  = val_or_par >
  < MAX  = val_or_par >
< PRED_FROM_LINK = linkFrom | linkFrom.sublink... >
< UPDATE_FROM_LINK = linkFrom | linkFrom.sublink... >
?connection?
  < BREAK_LEVEL = val_or_par >
  < MATCH_LEVEL = val_or_par >
```

Note that: INPUT, PRED_FROM_LINK, UPDATE_FROM_LINK, GLOBAL_TIME, GLOBAL_TIME_STEP qualifiers are mutually exclusive; only one of them may be specified in a LINK statement.

Here:

1. name : Link name.
2. "link_description" : Description , used by browsers.
3. INPUT : Input the variable created by this link, using link name as input variable name.
4. REPORT : Output the variable referenced by this link, using link name as report variable name.

5. VAL = val_or_par : Set the value of the variable defined by this link to a constant value 'val_or_par'. It assigns the constant value, as if it is input, to the variable defined by the LINK statement. This value can propagate to outside of this class if in the same link statement there are connection(s) to the port(s) of this class. This value can be overridden later by the INPUT or GLOBAL_TIME keywords referencing the same variable at higher levels.
6. INIT = val_or_par : Gives initial value to the variable. If the variable referenced by this link is a break variable the value is used only once, in the first Newton-Raphson iteration.
7. MIN, MAX : Give min, max value to the variable created by this link.
8. PRED_FROM_LINK : If the variable referenced by this link is a break variable, give initial value to it from the current value of linkFrom. Unlike the INIT keyword, PRED_FROM_LINK supplies the initial value for Newton-Raphson for every time step.
9. UPDATE_FROM_LINK : Makes the variable that is created by current link statement a previous value variable. Updating occurs at the beginning of the time step, prior to solving the system of equations. The value of the previous value variable remains the same during Newton-Raphson iterations.
10. GLOBAL_TIME : Connects the variable referenced by this link to calculation time value (t) that is specified by run control data.
11. GLOBAL_TIME_STEP :Connects the variable referenced by this link to calculation time step (dt) value that is specified by run control data.
12. ?connection? : This specifies either .port_of_this_class including the resolution of the subport if necessary e.g. .port_of_this_class .port_of_this_class.subport…, or connection to a port of an object declared in this class including the resolution of the subport,e.g.
    object.port
    object.port.subport...
13. BREAK_LEVEL : Break_level given to this connection.
14. MATCH_LEVEL : Match_level given to this connection.

4.14 INPUT Statement

The INPUT statement is exactly like the LINK statement with the INPUT keyword specified. Its is merely an alternative style.

4.15 EQUATIONS statement

The EQUATION statement specifies the equations that are used to generate the *C++* functions of this class. In future versions, this statement may be used by browsers and symbolic processors. It is a compound statement. An example is

```
EQUATIONS {
                p1.a = x ;
                p1.b = y ;
                p2   = z ;
                x = y^2 * z^2 , x > 0 ;
                BAD_INVERSES = y, z ;
}
```

Notes:

1. Currently, the parser does not use the Equations section. In future versions, the Functions section may be optionally generated from the Equations section.

2. In this example, x, y and z are "helper" symbols to simplify the equation. The notation p1.a means the a subport of port p1. In addition to the equation relating x, y, and z, we restrict x to positive values.
3. Currently, SPARK recognizes only one equation in an atomic class.

4.16 FUNCTIONS statement

The FUNCTIONS statement specifies the *C++* functions associated with the ports. It is a compound statement of the form

```
FUNCTIONS {
        port1 = inverse_fun1( port2, port3,... ) ;
        port2 = inverse_fun2( port1, port3,... )
                PRED = predictor_fun1( port1, port2, port3,...) ;
        port3 ;
}
```

Here *inverse_fun1* is the *C++* function that calculates the value of port1 from the values of all ports listed in (port2, port3,…). Similarly for *inverse_fun2*, where the PRED = construct is also specifies the *C++* function that calculates the predicted value of port2, as might be used in some types of numerical integration classes. If there is no *C++* function available for a port, either omit that port under FUNCTION, or give only the name of the port, e.g., port3 in the example.

4.17 Input From Files

SPARK does not distinguish between constant and time-varying boundary condition variables, i.e., inputs. All INPUTs (or LINKs with the INPUT keyword specified) will be sought from *.inp* files specified for the problem. To accommodate time varying inputs, the *.inp* file has the form

n	var_1	var_2	var_3	…	var_n
t_0	val_1	val_2	val_3	…	val_n
t_1	val_1	val_2	val_3	…	val_n
t_2	val_1	val_2	val_3	…	val_n
*					

Here var_k are the variable names defined as inputs and val_k are their values at times t_i. Constant values have the same value repeated at each time value. The final line with only * in it is optional, meaning that all values remain fixed from that point forward.

It is sometimes more convenient to use multiple input files, thus allowing different time stamp sequences. The input files are specified in the *InputFiles* clause of the *probName.run* file. See Section 3.4, page 43, for examples of when this might be useful. At run time, the SPARK solver opens each of the listed files, which are later searched when looking for input values.

Appendix A Using the HVAC Tool Kit

A.1 The SPARK HVAC Toolkit

The SPARK HVAC Toolkit is based on the ASHRAE Secondary Systems Toolkit (Brandemuehl 1993), supplemented with primary equipment models from DOE-2 (LBL 1984). This library of HVAC components is limited to steady state models. The models included are listed in Table A.1.

These classes are located in the *vspark\hvactk\class* directory, or in the *vspark\globalclass* directory if they are general in nature and thus apply to a wider range of problems. Each class has internal documentation in the form of a commented header. You should consult this header before using one of these classes. In addition, these headers are separately provided in rich text format (RTF) in the self-extracting *pkzip* file *rtflib.exe in vspark\bin*. You can examine these by executing this file with the class name you wish to see as an argument

```
rtflib cond.rtf <enter>
```

This will place the documentation for the *cond* class in the current working directory where it can be viewed with Microsoft Word, the free Microsoft Word Viewer available from Microsoft, or any other RTF viewer. These files are also provided in *pdf* format for viewing with the Adobe Acrobat Reader.

Many of these classes are lower-level macro or atomic classes from which the user level classes are built. These are automatically introduced into your problem as needed when you declare an object of the higher level class.

The SPARK classes in the *hvactk\class* directory are implemented using normal, atomic ports and links. Another version of the HVAC class library employs macro links and ports for the same set of classes where appropriate. These classes are in the *hvactkMP\class* directory.

A.2 Example Usage

Some examples of using these classes have already been seen in examples in this manual. For example, we used the *cond.cc* class in the *room_fc* problem in Section 2.7 (See page 28). In addition, every class has a test driver *.pr* file and associated *.inp* file in compressed form in *pr.exe* in *vspark\bin*. You can access one of these test drivers by executing *pr.exe* with the class name as an argument, e.g.,

```
pr cond.pr <enter>
pr cond.inp <enter>
```

This will place the driver problem and input files for *cond.cc* in the working directory. Alternatively, you can execute the provided batch file called *testhvac.bat* to extract, build, and execute the driver. First, you should go to the *vspark\hvactk* or other project directory. Then type:

```
testhvac cond <enter>
```

Results can be found in *cond.out.*

Note that the system models provided with the library show relatively complex macro classes that have been constructed from other Toolkit classes. These also have test drivers in the *pr.exe* compressed file.

Table A.1 SPARK HVAC Toolkit Classes

Class	**Description**
bf_ntu	Coil bypass factor vs. an Ntu-like parameter
bound	Bound a value
clipnorm	Bound a value between 0 and 1
capratel	Capacitance rate for water
cap_rate	Moist air capacitance rate
cclogic	Dry *vs.* wet coil decision logic
cond	Generic conductance relation
cpair	Specific heat of air
ctr1	Cooling tower Fr vs. range dependency
ctr2	Cooling tower Fr vs. approach dependency
diff	Difference
dxcap_m	Capacity variation with mass for DX AC unit
dxcap_t	DX AC unit capacity variation with outside dry and inside wet bulb temperatures
dxeir_m	EIR variation with mass flow rate
dxeir_t	DX AC unit EIR variation with TWb
effc1u	Ntu-effectiveness, stream 1 unmixed
effcbm	Ntu-effectiveness, cross flow both mixed
effcbu	Ntu-effectiveness, cross flow both unmixed
effctr	Ntu-effectiveness for counter flow
effncy	Forces two *input*s to *sum* to 1.0
effntu1	Exponential effectiveness vs. Ntu
effprl	Ntu-effectiveness for parallel flow
eintrp1	Exponential interpolation
eir1_oc	Curve fit for eir1 in open centrifugal compressor
eir2_oc	Curve fit for eir2 in DOE-2 open centrifugal compressor
enthalpy	Enthalpy, dry bulb, humidity relation.

enthvap	Enthalpy of water vapor
enthwat	Enthalpy of water
eq31	Equation 31 of ASHRAE HOF, Ch. 6
equal	Equality
fflp_blr	Boiler part load curve fit
fflp_dd	Fraction of full load power for discharge damper fan
fflp_iv	Fraction of full load power for inlet vane fan
fflp_vsd	Fraction of full load power for variable speed drive fan
htxeff	SPARK Heat exchanger effectiveness object
htxtemp	Temp vs. capacity flow vs. effectiveness
humratio	Humidity ratio vs. partial pressure of vapor
idealgas	Ideal gas law
indep_fr	Independent fractions
lat_rate	Latent heat rate object.
lintrp	Linear interpolation
lintrp1	Linear interpolation to 1
log10	SPARK log base-10 object.
max2	SPARK maximize object for two arguments
min2	SPARK minimum object for two arguments
neg	SPARK negation object
polyn3	3rd degree polynomial
poslim	Force to be positive
pow	SPARK exponentiation object.
propcont	Proportional controller
rcap_oc	Curve fit for capacity in open centrifugal compressor
rho	Moist air density vs. specific volume & humidity ratio
rhomoist	Moist air density vs. dry bulb and humidity ratio
safprod	SPARK safe product object
safquot	SPARK safe quotient object
safrecip	Safe reciprocal
satpress	Saturated pressure relationship for water.
satp_hw	Saturated Pressure (Hyland & Wexler)
satp_r	Saturated pressure of water vapor, residual method.
select	Logical if-then-else

sercond	Conductors in Series
square	Square of a value
sum	SPARK *sum* object
balance	Transport balance equation
dewpt	Dew point relationship for moist air using Walton's Saturation correlation
dewp_hw	Dew point using Hyland & Wexler saturation correlation.
enthsat	Dry bulb vs. enthalpy at saturation
relhum	Relative humidity
relh_hw	Relative humidity (Hyland & Wexler)
specvol	Specific volume of air
wetbulb	SPARK object defining the wet bulb temperature process.
wetb_hw	Wet Bulb (Hyland & Wexler)
gendiv	SPARK generic diverter object.
divsim	Diverter. Splits a flow stream into two streams.
mixer	Mixing box model for moist air
bf	Coil bypass ratio relationships
bf_adp	Bypass factor/apparatus dew point coil model
pumpsim	Simple pump
ctfunc	Cooling tower model correlation.
fann_dd	Discharge damper fan, mass flow-enthalpy interface
fann_iv	Inlet vane controlled fan, mass flow-enthalpy interface
fann_vsd	Variable speed drive fan, mass flow-enthalpy interface
fansim_n	Simple fan- part load coefficient & enthalpy/mass interface
fansim	Simple fan with part load coefficients in the interface
fan_dd	Discharge damper fan, volume flow -temperature interface
fan_iv	Inlet vane controlled fan, volume flow -temperature interface
fan_vsd	Variable speed drive fan, volume flow -temperature interface
htxc1u	Cross flow, stream 1 unmixed heat exchanger model
htxcbm	Cross flow, both streams mixed heat exchanger model
htxcbu	Cross flow, both streams unmixed heat exchanger model
htxctr	Counter flow heat exchanger model
htxprl	Parallel flow heat exchanger model
enthxc1u	Enthalpy exchanger model, cross flow, one stream unmixed
enthxcbm	Enthalpy exchanger model, cross flow, both streams mixed

enthxcbu	Enthalpy exchanger model, cross flow, both streams unmixed
enthxctr	Enthalpy exchanger model, counter flow
enthxprl	Enthalpy exchanger model, parallel flow
humeff	Humidity exchanger effectiveness
humex	Humidity exchanger model
drcc1u	Dry coil, cross flow, stream 1 unmixed
drccbm	Dry coil, cross flow, both streams mixed
drccbu	Dry coil, cross flow, both streams unmixed
drcctr	Dry coil, counter flow
drcprl	Dry coil, parallel flow
wcoilout	Wet Coil Leaving Conditions
wtcc1u	Wet Cooling/dehumidification Coil, cross flow, one stream unmixed
wtccbm	Wet Cooling/dehumidification Coil, cross flow, both streams mixed
wtccbu	Wet Cooling/dehumidification Coil, cross flow, both streams unmixed
wtcctr	Wet Cooling/dehumidification Coil, counter flow
wtcprl	Wet Cooling/dehumidification Coil, parallel flow
drywet	Dry/Wet Cooling Coil Model
indevap	Indirect evaporative cooler
tower	Cooling tower model
evaphum	Evaporative humidifier/cooler
airhx	Air to air heat exchanger
ccsim	Simple cooling coil model
acdx	Direct expansion air-conditioning unit model
econ	Economizer
boiler	Boiler
cchiller	DOE-2 single-stage compression chiller
vlvcirc	Flow circuit with non-linear valve and series flow resistance
zone	Simple steady-state zone model
vavsys	VAV System
zone_dd	Dual-duct controlled zone
ddhtbal	Dual-duct zone convergence enhancer
varmix	Variable mixing box
tstdhb	Test driver for ddhtbal
ddsys	Dual-duct (DD) System

cvrhsys	Constant volume reheat system
polyn3	3rd degree polynomial
bfd	Backward-forward difference integration object
room	Simple room with heat loss and air mass
bfd	Backward-forward difference integration object

Appendix B Preference Files

B.1 What are Preference Files?

Preferences file are external representations of objects of class *PrefList*. This C++ class is designed to allow storage and retrieval of (*key*, *value*) pairs, somewhat like a mapping. However, this class differs from a typical mapping in that it allows an hierarchical description of information. The example below will allow you to better understand the structure and format of SPARK preference files.

B.2 Uses of Preference Files in SPARK

Preference files are used several places in SPARK to store information about important aspects of the problem and how it is to be solved. For example, every SPARK problem has a *probName.prf* file that gives information about the problem component structure, and how each component is to be solved (See Section 3.10.1, page 53). Also, each problem has a run control file *probName.run* (See Section 3.14, page 65) with information about the simulation interval and other control issues. In some environments, a global *spark.prf* stores critical information about the SPARK installation. Here we explain the general format of all preference files.

B.3 Hierarchical Data

As an example, consider the need to store the description of a building. The building is to have a *Name*, a *Roof*, a *Floor*, and an arbitrary number of *Walls*. Although the *Name* has a simple string value, e.g., "MyBldg", *Roof*, *Floor* and every *Wall* has two attributes, *U* and *W*.

Figure B.1 shows this information as a general tree. It can also be thought of as an object called *theBuilding*. Every node in this tree can be viewed as a *key*, and the list of child nodes can be viewed as the *value* of that key. Thus *theBuilding* has a value which is the list *(Name, Roof, Walls, Floor)*, each of which is another tree. In turn, the root of each of these trees can be thought of as another key with its own value. The key *Name* has a single value, *myBldg*, and the key *Roof* has the value which is the list (*U, W*), each of which is a tree. The *U* and *W* keys at the roots of these trees each have a single value, (1.2) and (1.0) respectively. Note that nodes in the tree like *myBldg*, *1.2*, and *1.0* are distinctly different from nodes like *Name* or *Root* in that they have no children, i.e., they are leaves. Another way of saying this is that the "value" of a node like *myBldg* or *U* consist of an empty list ( ). These are the actual data stored in the structure. Note also that the path from the root to any leaf is a unique identifier of the data in the leaf. For example, *theBuilding.Roof.U* identifies the value 1.2.

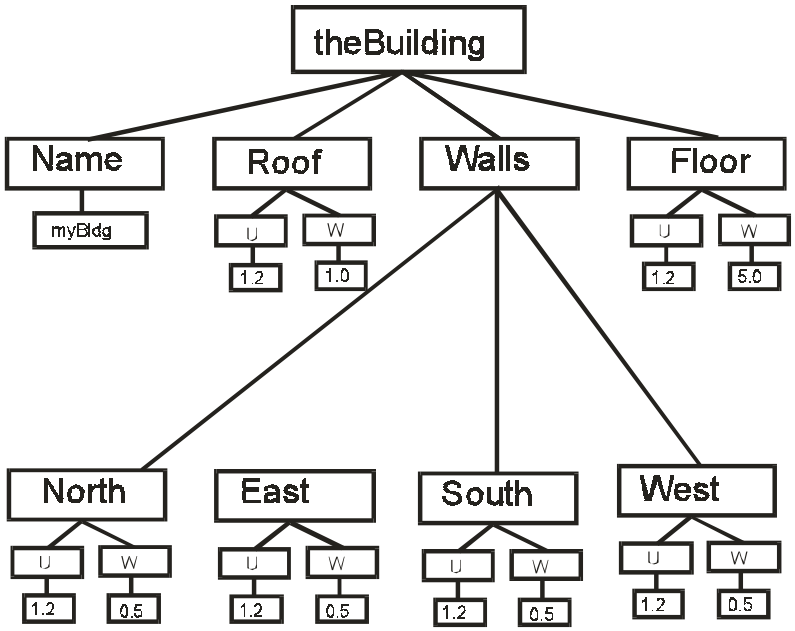

Figure B.1 Simple Building Represented as a Tree

B.4 Preference File for the Example

The preference file expresses this tree structure as text. The preference file for the tree in Figure B.1 is shown below.

The format follows the convention that a key is followed by a list representing its value, enclosed in parentheses. If the list is empty, indicated by empty parentheses, the implication is that the key is in fact actual data. Note that the key representing the file itself, in this case *theBuilding*, is not part of the stored data. This is because externally the operating system will know it by the assigned file name, and programs that read preference files assign the file contents, i.e., its value, to an instance of *prefItem* class. Consequently, it is not useful to store the name in the file itself, and the file content begins with an opening parenthesis, and ends with a closing parenthesis. With these conventions, here is the file for *theBuilding*:

```
(
      Name (myBldg ( )
      )
      Roof (
            U (1.2 ( )
            )
            W (1.0  ( )
            )
      )
      Walls (
            North (
                  U (1.2 ( )
                  )
                  W (0.5 ( )
                  )
            )
            South (
```

```
                  U (1.2 ( )
                  )
                  W (0.5 ( )
                  )
            )
            East (
                  U (1.2 ( )
                  )
                  W (0.5 ( )
                  )
            )
            West (
                  U (1.2 ( )
                  )
                  W (0.5 ( )
                  )
            )
      )
      Floor (
            U (1.2 ( )
            )
            W (5.0  ( )
            )
      )
)
```

Since *theBuilding* tree has four first-level nodes, between file opening and closing parenthesis there are four main clauses, each consisting of a key followed by a parenthetic expression representing the value of the key. The first-level keys are the nodes in the tree, *Name*, *Roof*, *Walls*, and *Floor*. The *Name* key has a simple value, the building name string “myBldg”, so it is followed by a empty parentheses. Note that the format is delimited entirely by the parentheses so spaces in strings are allowed, and no quoting is necessary. The *Roof* and *Floor* keys have values that are trees with nodes representing *U* and *W*. The *U* and *W* keys have simple values, so they are followed by empty parentheses. The *Walls* identifier has a more complex structure, namely four trees, each with a structure like *Roof* and *Floor*.

References

Anderson, J. L. (1986). *A Network Language for Definition and Solution of Simulation Problems*, Lawrence Berkeley Laboratory.

Brandemuehl, M. J. (1993). *HVAC 2 Toolkit: A Toolkit for Secondary HVAC System Energy Calculations*, Joint Center for Energy Management, University of Colorado.

Buhl, W. F., A. E. Erdem, et al. (1993). "Recent Improvements in SPARK: Strong Component Decomposition, Multivalued Objects, and Graphical Interface." *Proceedings of Building Simulation '93*, Adelaide, International Building Performance Simulation Association. Available from Soc. for Computer Simulation International, San Diego, CA.

Char, B. W., K. O. Geddes, et al. (1985). *First leaves: a tutorial introduction to Maple, in Maple User's Guide*. Waterloo, Ontario, WATCOM Publications Ltd.

Conte, S. D. and C. de Boor (1985). *Elementary Numerical Analysis: An Algorithmic Approach*. McGraw-Hill Publishing Co.

Dennis, J. E. and Schnabel, R. B. (1996). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Classics in Applied Mathematics 16, SIAM.

LBL (1984). *DOE-2 Reference Manual*, Lawrence Berkeley Laboratory.

McHugh, J. (1990). *Algorithmic Graph Theory*. Englewood Cliffs NJ 07632, Prentice Hall.

Nataf, J.-M. and F. C. Winkelmann (1992*). Automatic Code Generation in SPARK: Applications of Computer Algebra and Compiler-compilers*. Berkeley, CA, Simulation Research Group, Lawrence Berkeley Laboratory.

Nierstrasz, O. (1989). "Survey of Object-Oriented Concepts." *Object-Oriented Concepts, Databases, and Applications*. W. Kim and F. H. Lochovsky. New York/Reading, ACM Press/Addison-Wesley**:** 3-21.

Press, W. H., B. P. Flannery, et al. (1988). *Numerical Recipes in C*. Cambridge, Cambridge University Press.

Rand, R. H. (1984). *Computer Algebra in Applied Mathematics: An Introduction to MACSYMA*. Boston.

Sahlin, P. and E. F. Sowell (1989). "A Neutral Format for Building Simulation Models." *Proceedings of Building Simulation '89*, Vancouver, BC, International Building Performance Simulation Association.

Sen, W. T. (1994). *untitled draft.* Singapore.

Sowell, E. F. and W. F. Buhl (1988). “Dynamic Extension of the Simulation Problem Analysis Kernel (SPANK).” *Proceedings of the USER-1 Building Simulation Conference*, Ostend, Belgium, Soc. for Computer Simulation International.

Sowell, E. F., K. Taghavi, et al. (1984). “Generation of Building Energy System Models.” ASHRAE Trans. **90**(Pt. 1): 573-86.

Glossary of Terms

Algorithmic programming

A sequence of operations and assignments leading from prescribed inputs to prescribed outputs.

Assignment

In computer languages, assignment is the action whereby a value is associated with an identifier representing a variable. Although the symbol "=" is often used for assignment, e.g., X = 2*y, assignment is different from mathematical equality because the latter implies that the expressions at the left and right of the "=" symbol are always equal. In particular, a sequence of assignments are order dependent, while a set of mathematical equations are not. See algorithmic programming.

Atomic classes

A model comprising a single equation with used variables linked to its ports. Acts as a template for instantiation of atomic objects.

Break level

An integer 0-10 expressing the desirability of using the associated link to break cycles in the computation graph.

Class

A general description of an equation (atomic class) or group of related equations (macro class). A class acts as a template for instantiation of objects.

Continuous variable

Variable that can take on any real value within a range.

Cut set

A set of variables (links) that will break all cycles in the computation graph. SPARK attempts to minimize the cut set size. The associated variables are called "break variables" and are used for iterative solution.

Differential algebraic equation system (DAES)

A system of differential and algebraic equations for simultaneous solution.

Discrete state variable

A variable that can take on only specific values rather than any real value within a range.

Dynamic variables

A variable for which the derivative appears in a differential equation.

Graph

See mathematical graphs.

Ill posed

A problem that is not well posed is said to be ill posed. See Well posed.

Implicit inverse

A form of an equation in which a particular variable is on the left, but also occurs in the right side expression. Used when explicit inverses cannot be obtained. Solution requires iteration.

Initialization

Value of variable at *InitialTime*. Required for dynamic variables, and for break variables.

InitialTime

The time when simulation starts. That is, the time at which initial conditions for differential equations apply.

Input/output free

A style of model expression which provides a set of equations rather than an algorithm. Since any set of inputs that leads to a well posed problem can be specified in conjunction with these equations, it is sometimes called input/output free.

Instantiate

To create an object instance based on the class definition. The *declare* keyword performs instantiation in SPARK.

Integration formula

A formula used in numerical solution of differential equations to calculate a value for the integration variable at the next point in time. Can be explicit, in which the new value appears only on the left, or implicit in which case the new value and or the new derivative appears also in the right side expression.

Interface variable

A variable defined in a class that is to be visible from outside. Interface variables are defined with the *port* keyword.

Inverse

Precisely, a form of an equation in which a particular variable is isolated on one side; i.e., a formula for a variable. In SPARK, we use the term explicit inverse for such a formula. See also Implicit inverse.

Jacobian

Square matrix of partial derivatives of residual equations with respect to the break variables in a strongly connected component.

Macro classes

A group of SPARK atomic or other macro classes linked together through their respective ports to form a subsystem model. A macro class can be use wherever an atomic class can be used.

Match level

An integer 0-10 expressing the desirability of matching the associated link variable with the associated object port.

Mathematical graphs

A structure comprising a set of vertices (nodes) and edges (arcs) which connect them. Often used to model systems of interacting entities.

Object oriented

Modeling methodology in which the model behavior and data are encapsulated in a model entity comparable to the physical entity that it represents. Communicates with other parts of the model only through its interface ports.

Parser

The program that interprets the SPARK input files as the first step toward solution.

Prediction

Value of break variable at beginning of iterative solution. Defaults to value at previous time step if not specified as *pred_from_link*.

Propagation

Process by which SPARK infers certain link or port statement settings, e.g., *init*, *max* and *min*, from settings at lower or higher levels with respect to macro classes and problem specifications.

Relaxation coefficient

Multiplier, usually a fraction, on calculated correction that is actually applied in order to get new break variable values during iterative solution.

Retained state

Value that needs to be saved between successive uses of an object. Currently, SPARK objects cannot retain state internally. However, values of link variables are retained for 4 previous time steps. State can also be retained through use of the *update_from_link* concept.

Solver

The executable program that SPARK builds to solve a particular problem. Called *probName.exe* (Windows) or *probName* (UNIX). The underlying programs used by SPARK in constructing executable are also referred to as “the solver” in places.

Strong component

Short for strongly connected component. In graph theory, a maximal set of vertices and edges that allow any vertex in the set to be reached from every other vertex. In SPARK, corresponds to a separately solvable sub-problem, discovered automatically.

Symbolic manipulation

Operations on mathematical expressions in terms of contained symbols, as opposed to numerical evaluation. The goal might be solution for one or more symbols in terms of the others. Often done with computer software, i.e., computer algebra.

Updating

Setting value of Previous Value Variable to the previous value of variable specified with the *update_from_link* keyword. Occurs at beginning of time step, before solving the components.

Well posed

A problem is said to be well posed if it admits at least one solution. One requirement is an equal number of equations (objects) and unknowns (links). There also must be a complete matching, i.e., a matching of each

variable to a unique equation inverse. However, problems can meet these requirements and still not be well posed. For example, y=f(x) and y = g(x) may not intersect.

Index

Notice

This work was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Building Technology, State and Community Programs, Office of Building Systems of the U.S. Department of Energy, under contract DE-AC03-76SF00098.